Bachelorarbeit

Entwicklung eines Intel HD Audio Soundkartentreibers in Rust

vorgelegt von

Sebastian Heidelberg

aus Düsseldorf

Abteilung Betriebssysteme
Prof. Dr. Michael Schöttner
Heinrich-Heine-Universität Düsseldorf

18. Mai 2024

Erstgutachter:	Prof. Dr. Michael Schöttner
Zweitgutachter:	Prof. Dr. Martin Mauve
Betreuer:	Dr. Fabian Ruhland

Inhaltsverzeichnis

Kapitel 1

Einleitung

Jedes an einen Computer angeschlossene Gerät ist spezialisiert auf eine Aufgabe. Beispielsweise dienen Tastatur und Maus der Eingabe von Befehlen, eine Netzwerkkarte verbindet den Computer mit anderen Systemen und eine Soundkarte übersetzt Audiosignale aus der Sprache des Computers in die Sprache eines an diese Soundkarte angeschlossenen akustischen Geräts und umgekehrt. Das Betriebssystem eines Computers verwaltet die verfügbaren Hardware-Ressourcen wie Tastatur, Maus, Netzwerk- oder Soundkarte und macht sie Anwendungsprogrammen zugänglich. Zu diesem Zweck benötigt es die Unterstützung von spezialisierter Software, sogenannten Gerätetreibern oder kurz Treibern. Ein Treiber ermöglicht es dem Betriebssystem, ein bestimmtes Gerät zu erkennen und stellt diesem Betriebssystem ein Application Programming Interface (API) zur Verfügung, über welches es mit dem Gerät kommunizieren und dieses steuern kann. Da der Treiber für die Erfüllung seiner Aufgaben wiederum die vom Betriebssystem bereitgestellten Ressourcen nutzt und diese von Betriebssystem zu Betriebssystem unterschiedlich sind, ist ein Treiber nicht nur geräteabhängig, sondern auch betriebssystemspezifisch.[1]
Folglich benötigt einerseits jedes neu entwickelte Betriebssystem seine eigenen neuen Treiber für bereits existierende Geräte und andererseits muss ebenfalls ein neuer Treiber geschrieben werden, wenn ein bereits existierendes Betriebssystem ein neues Gerät unterstützen soll. Die Entwicklung von Gerätetreibern stellt also einen essenziellen Bestandteil der Betriebssystemprogrammierung dar.

1.1 Zielsetzung

Seit 2023 wird am Lehrstuhl für Betriebssysteme an der Heinrich-Heine-Universität Düsseldorf ein in der Programmiersprache Rust geschriebenes Betriebssystem mit dem Namen Distributed Data-Driven OS (D3OS) entwickelt, welches zum Entstehungszeitpunkt dieser Arbeit zu Beginn des Jahres 2024 unter anderem bereits einen Treiber für eine Tastatur, jedoch noch keinen Treiber für eine Soundkarte besitzt.
Das Ziel dieser Arbeit ist die Entwicklung eines Sounkartentreibers für das D3OS. Dieser Treiber soll dem Betriebssystem eine API bereitstellen, über welche es eine nach der Intel HD Audio Spezifikation[2] gebaute Soundkarte bedienen kann. Das D3OS soll mittels dieser API ein im Systemspeicher definiertes digitales Audiosignal korrekt über einen analogen Ausgang der Soundkarte bzw. einen an diesen Ausgang angeschlossenen Lautsprecher wiedergeben können.

1.2 Aufbau der Arbeit

Nachdem in 1 in das Thema dieser Bachelorarbeit eingeführt und eine Zielsetzung formuliert wurde, werden in 2 die für die Entwicklung des Treibers benötigten Technologien vorgestellt und für das Verständnis der Arbeit wichtige Begriffe und Konzepte erklärt. Anschließend wird in 3 die konkrete Implementierung des entwickelten Treibers beschrieben. In 4 wird zunächst analysiert, welche Herausforderungen während der Treiberentwicklung bewältigt werden mussten, bevor überprüft wird, inwiefern der Treiber die an ihn gestellten Anforderungen erfüllt. Zuletzt werden die vorgestellten Konzepte, die Umsetzung dieser Konzepte durch die Implementierung sowie die Ergebnisse der Evaluation in 5 zusammengefasst und ein Ausblick auf mögliche Weiterentwicklungen gegeben.

1.3 Umgang mit Begriffen

Um sowohl das Lesen dieser Arbeit als auch die Zuordnung von Begriffen aus dem Text zu Begriffen aus der Intel HD Audio Spezifikation sowie zu Begriffen aus dem Code des entwickelten Treibers zu erleichtern, wurden einerseits einige Begriffe im Text besonders hervorgehoben und andererseits Begriffe aus der in Englisch verfassten Spezifikation nicht übersetzt.
Alle Begriffe, welche ein konkretes in der Spezifikation definiertes Konzept repräsentieren, wurden *kursiv* gesetzt. Alle Begriffe, welche eine konkrete Struktur aus dem Code des Treibers repräsentieren, wurden in `Schreibmaschinenschrift` gesetzt. Mit dem *Controller* ist also das Konzept aus der Spezifikation gemeint und mit dem `Controller` dessen Repräsentation im Code.

Kapitel 2

Grundlagen

Beim Verfassen dieser Arbeit wurde davon ausgegangen, dass dieser Text über die Entwicklung eines Treibers für ein Forschungsbetriebssystem hauptsächlich von Personen gelesen wird, welche sich zuvor bereits zumindest oberflächlich mit den Grundlagen der Betriebssystemprogrammierung beschäftigt haben. Aus diesem Grund werden die folgenden Begriffe in den nächsten Kapiteln ohne weitere Erläuterungen verwendet und es wird darum gebeten, unklare Begriffe an anderer Stelle nachzuschlagen:

- Betriebssystem, Kernel, Prozess und Thread
- Virtuelle Speicherverwaltung, Paging und das Abbilden (Mapping) von Adressen zwischen Adressräumen
- Heap, Speicherallokation, Zeiger auf Speicheradressen und Garbage Collection
- Arbeitsspeicher und Cache
- Memory-Mapped I/O (MMIO), Direct Memory Access (DMA) und Bus Mastering
- Bus zur Datenübertragung und Serialisierung von Daten
- Register und Offset eines Registers
- Basic Input/Output System (BIOS), Initialisierung und Enumerierung eines Geräts
- Hardware-Interrupts, Interrupt-Controller und Interrupt-Handler
- Application Programming Interface (API)
- Buffer, inbesondere Ringbuffer

Der Treiber wurde zwar in Rust entwickelt, jedoch steht nicht die genutzte Sprache im Vordergrund dieser Arbeit, sondern die Umsetzung der durch die Intel HD Audio Spezifikation festgelegten Vorgaben zur Implementierung eines Soundkartentreibers. Also wurde beim Verfassen dieser Arbeit darauf geachtet, nicht zu tief in die Besonderheiten der Sprache einzutauchen und Code-Beispiele wurden so ausgewählt, dass sie prinzipiell auch als Pseudocode gelesen werden können, sodass Vorkenntnisse in Rust für das Verständnis dieser Arbeit nicht zwingend notwendig sind. Gleichzeitig nutzt Rust viele Konzepte, welche in ähnlicher Form auch aus anderen Programmiersprachen, wie beispielsweise C oder Java, bekannt sind. Deshalb werden auch folgende Begriffe im Verlauf dieser Arbeit unerläutert benutzt:

- Struct und Enum sowie Definition, Instanziierung und Konstruktor-Funktion eines Structs bzw. Enums
- Funktion und Methode sowie Definition, Aufruf, Parameter und Rückgabewert einer Funktion bzw. Methode
- Code-Abhängigkeiten (Dependencies)
- Sichtbarkeit (öffentlich und privat)
- Trait
- Statische Variable und Referenz auf ein Objekt

Im Folgenden werden nun die wichtigsten Begriffe und Konzepte für das Verständnis dieser Arbeit sowie die während der Treiberentwicklung genutzten Technologien vorgestellt. Der Fokus liegt dabei auf dem in 2.6 vorgestellten Intel HD Audio.

2.1 Die Programmiersprache Rust

Die erste Version von Rust wurde am 15.05.2015 veröffentlicht[3], sodass die Sprache zum Zeitpunkt der Entstehung dieser Arbeit erst etwa neun Jahre alt ist. Obwohl sie also noch sehr jung ist und stetig durch die aktive Rust Community[4] weiterentwickelt und verändert wird, nutzen zahlreiche große Unternehmen Rust bereits in Produktion[5].
Rust erlaubt es, wie auch die zu Beginn des 21. Jahrhunderts vorherrschende Systemprogrammiersprache C, hardwarenah zu Programmieren und beispielsweise manuell den Speicher zu verwalten. Allerdings muss allozierter Speicher in Rust im Gegensatz zu C und ähnlich wie in Sprachen mit einem Garbage Collector, wie beispielsweise Java, nicht manuell wieder freigegeben werden[1], sodass eine ganze Familie von für C-Programme typischen Fehlerquellen und Sicherheitslücken[7] in Rust gar nicht erst existiert. Zusätzlich bietet Rust, anders als C, einige komfortable Abstraktionen, wie beispielsweise Traits, welche ebenfalls eher aus höheren Programmiersprachen wie Java bekannt sind, ohne dabei an Geschwindigkeit einbüßen zu müssen.[8] Der Erfinder von Rust, Mozilla Research Mitarbeiter Graydon Hoare, antwortete in einem Interview aus dem Jahr 2012 auf die

[1] Dafür nutzt Rust jedoch keinen Garbage Collector, sondern ein sehr idiomatisches Konzept namens Ownership.[6]

provokante Frage danach, was Rust besser als C mache:
„Primarily, it's just much safer, less likely to crash. Your code really has to mean it, if it's going to do something memory-unsafe. And we don't tax you too much for that memory safety, unlike fully garbage-collected systems. Various other conveniences are also worth noting: the memory model translates to a safe concurrency model, and there's good support for the usual modern conveniences like closures, traits, namespaces, destructors, Unicode, type inference, immutable memory, disjoint unions, etc.“[9]

2.1.1 Unsicheres Rust

Der Rust Compiler überprüft ein Programm auf dessen Speichersicherheit und lässt Ausdrücke, welche diese Sicherheit verletzen können, nur innerhalb von Code-Blöcken oder Funktionen zu, die mit dem Keyword `unsafe` markiert wurden.[10] Solche Code-Abschnitte werden unsicheres Rust genannt. Ein Beispiel für einen Ausdruck, welcher nur in unsicherem Rust erlaubt ist, ist die Dereferenzierung eines Zeigers, um auf eine beliebige Adresse im Speicher zuzugreifen.[11] Solch ein Speicherzugriff ist inhärent unsicher, da sein Gelingen immer erst zur Laufzeit überprüft werden kann. Der entwickelte Treiber dereferenziert Zeiger, um dadurch auf verschiedene von der Soundkarte genutzte DMA-Bereiche im Arbeitsspeicher zuzugreifen und so mit der Soundkarte zu kommunizieren. Für diese Zugriffe muss also unsicheres Rust verwendet werden.
Da der Compiler bei unsicherem Rust nicht bei der Aufrechterhaltung der Speichersicherheit helfen kann, sollte ein Programm so wenig Gebrauch wie möglich von unsicherem Rust machen. Außerdem sollten unsichere Code-Blöcke in Funktionen gekapselt werden, welche selbst wiederum sicher sind und somit eine sichere API zur Ausführung des unsicheren Codes bereitstellen.[12]

2.1.2 Paket-Management mit Cargo

Große Projekte werden in Rust durch Packages, Crates und Module strukturiert.[13] Ein Package besteht aus mindestens einer Crate und eine Crate aus mindestens einem Modul. Cargo ist Rusts Build-System und Paket-Manager und wird von der großen Mehrheit aller Rust-Projekte genutzt.[14] Cargo überprüft, ob alle für das Bauen eines Projekts benötigten Crates in der korrekten Version vorhanden sind und lädt fehlende Abhängigkeiten automatisch herunter. Ergänzend dazu erlaubt der Task-Runner cargo-make[15], zusätzliche Befehle und Skripte im Kontext des Build-Prozesses auszuführen.
In 6.1 wird erklärt, wie das D3OS mithilfe von Cargo und cargo-make gebaut werden kann.

2.2 Das Betriebssystem Distributed Data-Driven OS (D3OS)

Seit 2023 wird in der Abteilung Betriebssysteme der Heinrich-Heine-Universität Düsseldorf (HHU) ein eigenes verteiltes Forschungsbetriebssystem in Rust entwickelt: das Distributed Data-Driven OS (D3OS).[16] Es basiert auf dem ebenfalls in Rust geschriebenen Blog OS, welches von Philipp Oppermann seit 2018 in Form eines Internetblogs veröffentlicht und zum Zeitpunkt der Fertigstellung dieser Arbeit aktiv weiterentwickelt wird.[17]

Der Code des D3OS befindet sich in einem öffentlich zugänglichen, von der Abteilung Betriebssysteme der HHU verwalteten Repository auf GitHub.[18] In seinem frühen Entwicklungsstand bietet das D3OS fundamentale Betriebssystemfunktionalität wie Paging, Interrupt Handling oder die Bereitstellung eines Heaps und besitzt bereits einige Gerätetreiber, unter anderem für eine PS/2-Tastatur. Am Ende des Startvorgangs öffnet sich ein Terminal-Fenster, in dem das Programm `hello` ausgeführt werden kann, welches die Nummern des eigenen Threads und Prozesses ausgibt:

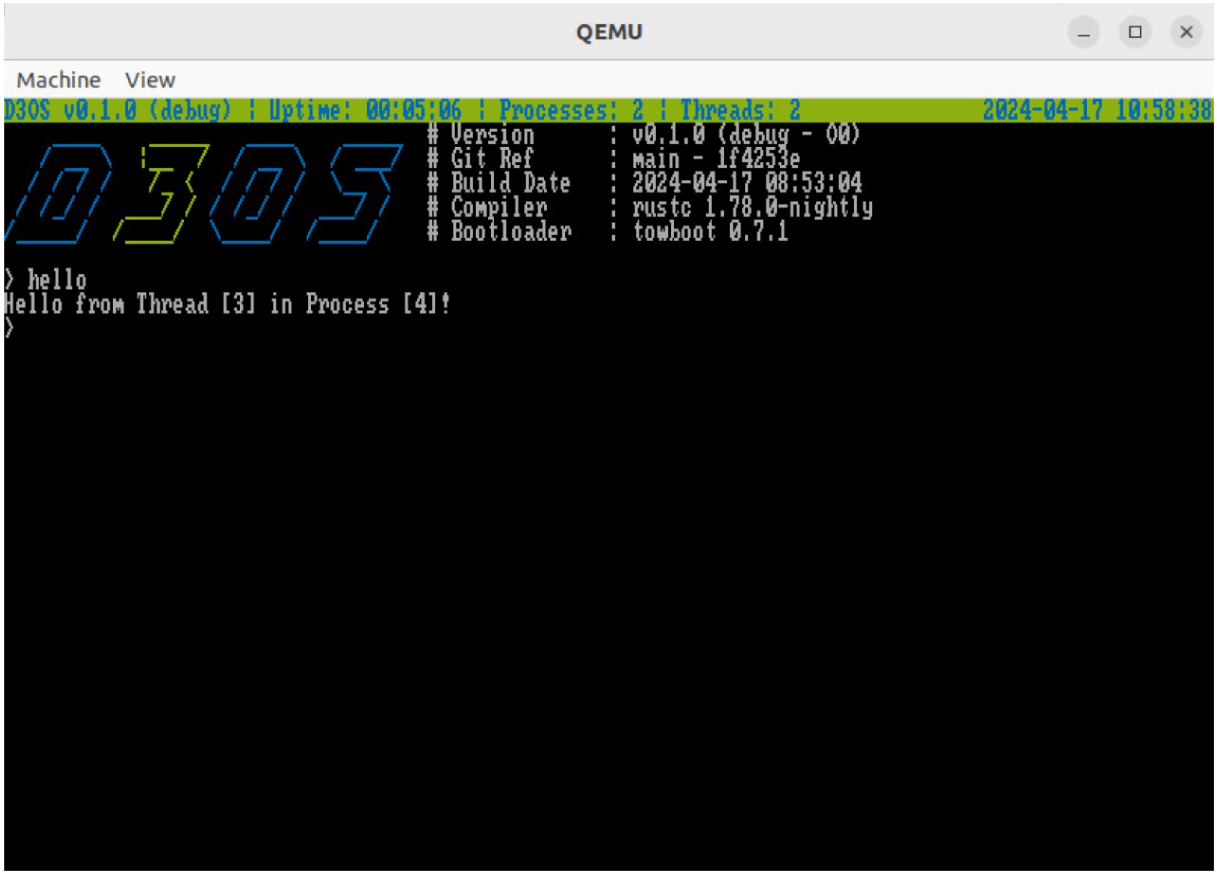


Abbildung 2.1: Das D3OS nach Ausführung des Programms `hello`

2.2.1 Struktur des D3OS

Das gesamte Betriebssystem bildet ein Package[2], der Kernel des D3OS[3] eine eigene Crate innerhalb dieses Packages und der bereits angesprochene Treiber für eine PS/2-Tastatur[4] ein Modul innerhalb dieser Crate. Im Rahmen der Implementierung des Treibers wurde diese Crate um insgesamt vier Module erweitert.

[2]Pfad im Repository: D3OS/os
[3]Pfad im Repository: D3OS/os/kernel
[4]Pfad im Repository: D3OS/os/kernel/src/device/ps2.rs

2.2.2 Virtuelle Speicherverwaltung und Direct Memory Access (DMA)

Das D3OS besitzt eine virtuelle Speicherverwaltung und somit verschiedene Adressräume. Die beiden für das Verständnis dieser Arbeit wichtigen Adressräume sind der physische Adressraum und der virtuelle Kernel-Adressraum. Die Soundkarte kennt ausschließlich physische Adressen, auf welche sie über DMA zugreift, und der Treiber ausschließlich virtuelle Adressen im Kernel-Adressraum. Damit Soundkarte und Treiber miteinander kommunizieren können, muss eine Verbindung zwischen den beiden Adressräumen hergestellt werden, indem die von der Soundkarte genutzten physischen Adressen auf virtuelle Adressen im Kernel-Adressraum abgebildet werden.

2.3 Entwicklungsumgebung und Workflow

Während der Treiberentwicklung wurden insgesamt drei Rechner benutzt: ein Entwicklungsrechner, auf welchem der Treiber geschrieben wurde, und zwei Testrechner, auf welchen der geschriebene Code getestet wurde. Der erste Testrechner war dabei ganz klassisch ein physischer Rechner mit integrierter Soundkarte und der zweite ein virtualisierter Rechner, welcher mithilfe der freien Open-Source-Virtualisierungssoftware QEMU[19] auf dem Entwicklungsrechner emuliert wurde. In einem repetitiven Prozess wurde immer zunächst etwas Code auf dem Entwicklungsrechner geschrieben und unmittelbar anschließend getestet, indem das modifizierte D3OS kompiliert und auf einem der beiden Testrechner hochgefahren wurde. Das um den fertige Treiber erweiterte D3OS befindet sich ebenfalls in einem öffentlichen Repository auf GitHub.[20] Wie das D3OS kompiliert werden kann, wird in 6.1 beschrieben.
Das Testen in der virtuellen Umgebung war sowohl zeitsparender, unkomplizierter als auch komfortabler als das Testen auf physischer Hardware, da für den Start des D3OS auf dem virtualisierten Rechner kein bootbarer USB-Stick erstellt werden musste und außerdem ein Debugger genutzt werden konnte. Jedoch funktionierte die Emulierung der Soundkarte in QEMU auf dem Entwicklungsrechner nur teilweise, sodass insbesondere in der zweiten Hälfte der Entwicklungsphase vornehmlich auf dem physischen Rechner getestet werden musste. In 4.1 wird das inkorrekte Verhalten der virtuellen Soundkarte näher beschrieben.

2.4 Peripheral Component Interconnect (PCI)

Über den PCI-Bus eines Rechners können Daten zwischen dem Rechner und mit dem PCI-Bus verbundenen Geräten ausgetauscht werden.[21] Jedes an den PCI-Bus angeschlossene Gerät besitzt seinen eigenen PCI Configuration Space[22] und darin enthaltene Status- und Kontrollregister, über welche dieses Gerät identifiziert, initialisiert und konfiguriert werden kann. Wie Initialisierung und Konfigurierung durchzuführen sind, ist von Gerät zu Gerät unterschiedlich und wird für eine Intel HD Audio Soundkarte in 2.6.2.1 beschrieben.

2.5 Analoge und digitale Audiosignale

Eine Intel HD Audio Soundkarte benutzt das Standardverfahren zur Digitalisierung von analogen Audiosignalen, die Pulse-Code-Modulation, bei der ein zeitkontinuierliches analoges Signal mit einer in der Regel festen Abtastrate in ein zeitdiskretes digitales Signal umgewandelt wird.[23] Das elektrische Bauteil, welches die Umwandlung eines analogen Signals in ein digitales vornimmt, wird Analog-Digital-Konverter (ADC[5]) genannt und das Bauteil, welches in umgekehrter Richtung ein digitales in ein analoges Signal transformiert, heißt Digital-Analog-Konverter (DAC[6]). Für die Begriffe Abtastwert und Abtastrate werden im Rahmen dieser Arbeit die ebenfalls geläufigen Begriffe Sample bzw. Samplerate benutzt. Die Kardinalität des Wertebereichs der Abtastung wird als Bittiefe bezeichnet.

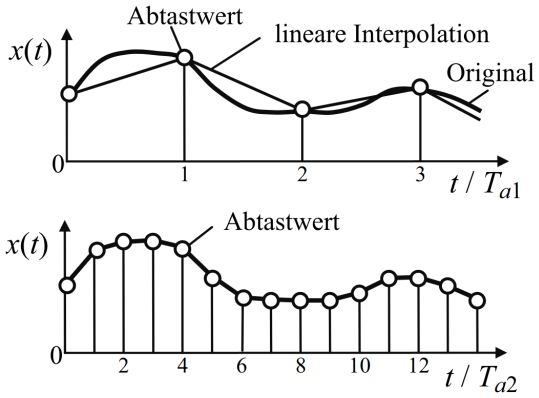
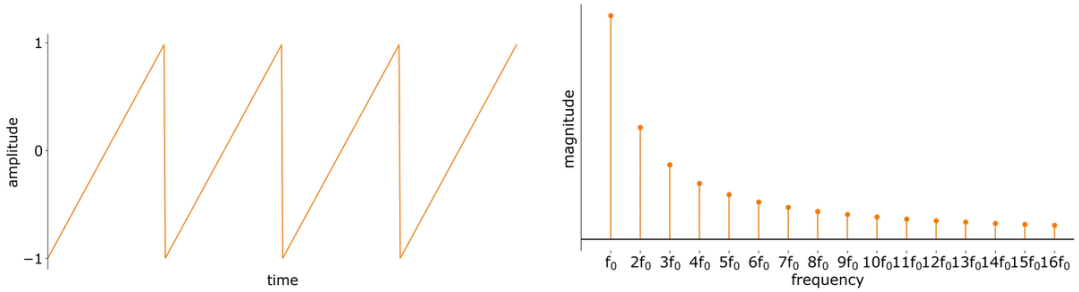


Abbildung 2.2: Abtastung eines analogen Signals[24]

Für die Evaluation des Treibers in 4.2 wurde ein digitales Signal in Form einer Sägezahnschwingung mit konstanter Grundfrequenz in den Speicher gelegt. Eine Sägezahnschwingung besitzt ein charakteristisches Obertonspektrum mit Spitzen bei allen natürlichen Vielfachen der Grundfrequenz, wobei die Amplituden dieser Spitzen streng monoton fallend sind.



(a) Wellenform (b) Obertonspektrum

Abbildung 2.3: Sägezahnschwingung[25]

[5]Dies ist die gebräuchliche Abkürzung der englischen Bezeichnung analog-to-digital converter.
[6]Dies ist die gebräuchliche Abkürzung der englischen Bezeichnung digital-to-analog converter.

2.6 Intel HD Audio

In der Spezifikation zu Intel HD Audio (IHDA) werden sowohl die Architektur als auch das Programmiermodell einer IHDA-Soundkarte definiert. Die Spezifikation richtet sich somit einerseits an Hersteller von Hardware und andererseits an Treiberentwickler, wie den Autor dieser Arbeit. Die aktuellste Revision der IHDA-Spezifikation ist Revision 1.0a vom 17.06.2010.[2] Auf dieser Revision basiert auch die Entwicklung des im Rahmen dieser Arbeit entstandenen Treibers.

2.6.1 Architektur

Eine IHDA-Soundkarte besteht aus drei Hardware-Bausteinen: genau einem *Controller*, genau einem *Link* und mindestens[7] einem *Codec*.[26]
Der *Controller* besitzt mehrere DMA-Engines, über die er zu verschiedenen Zwecken auf den Systemspeicher zugreift und stellt somit die nach innen gerichtete Schnittstelle der Soundkarte zum Rechner dar. Auf der anderen Seite werden akustische Geräte wie Lautsprecher oder Mikrofone an den *Codec* angeschlossen, sodass der *Codec* die Schnittstelle der Soundkarte zur Außenwelt bildet. Der *Link* bildet die physische Verbindung zwischen *Controller* und *Codec*, über welche digitale Audiosignale in beide Richtungen übertragen werden können.

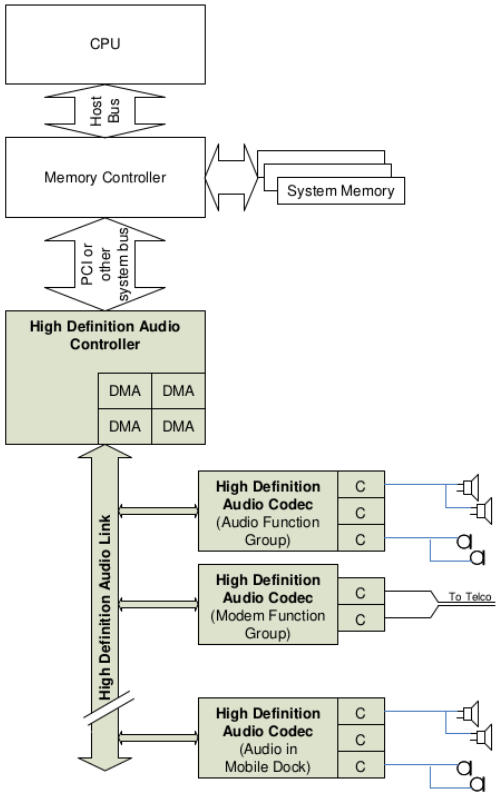


Abbildung 2.4: Blockdiagramm der Hardware-Architektur[27]

[7] Sowohl die virtualisierte Soundkarte in QEMU als auch die integrierte Soundkarte des für die Entwicklung genutzten Testrechners besaßen genau einen *Codec*. Aus diesem Grund wird im Rest dieser Arbeit meist nur von dem *Codec* und nicht von den *Codecs* gesprochen.

Vereinfachend veranschaulicht bildet der *Controller* das Gehirn der Soundkarte, während die *Codecs* zugleich Mund und Ohren entsprechen. Von den Ohren Gehörtes und durch den Mund zu Sprechendes wird über den *Link* als spezialisierten bidirektionalen Nervenstrang von den Ohren zum Gehirn bzw. vom Gehirn zum Mund übertragen.

Um ein besseres Verständnis über die Funktion jedes dieser drei Bausteine zu ermöglichen, werden sie im Folgenden jeweils im Detail betrachtet.

2.6.1.1 Die drei Hardware-Bausteine: Controller, Link und Codec

Controller: Der *Controller* ist ein Peripheriegerät, welches in der Regel über PCI, gegebenenfalls aber auch über eine andere Host-Schnittstelle mit dem Systemspeicher verbunden ist.[26] Im Kontext dieser Arbeit wird jedoch ausschließlich der Regelfall betrachtet. Der *Controller* nutzt als Bus Master seinen eigenen DMA-Controller („First-Party-DMA“) und kann somit ohne die Unterstützung eines externen DMA-Controllers und der CPU („Third-Party-DMA“) direkt in den Systemspeicher schreiben und aus dem Systemspeicher lesen. Er besitzt eine oder mehrere DMA-Engines, welche so konfiguriert werden können, dass sie ein digitales Audiosignal über den *Link* entweder vom Systemspeicher zu einem *Codec* für die Wiedergabe eines Audiosignals oder aber von einem *Codec* zum Systemspeicher für die Aufzeichnung eines Audiosignals übertragen[8]. Neben der Übertragung der eigentlichen Audiodaten nutzt der *Controller* DMA zu verschiedenen weiteren Zwecken, welche nach und nach in den folgenden Abschnitten vorgestellt und am Ende dieses Grundlagen-Kapitels noch einmal zusammengefasst werden.

Treiber und *Controller* kommunizieren miteinander über die *Registers*[28] des *Controllers*, welche bei der Initialisierung und Enumerierung des PCI-Busses während des Betriebssystemstarts vom BIOS an eine feste Adresse im physischen Adressraum abgebildet werden, die sogenannte MMIO-Basisadresse[9]. Die *Register* stellen die zentrale Programmierschnittstelle dar, über die der Treiber sowohl statische als auch dynamische Informationen über die Soundkarte einholt, den *Controller* und dessen DMA-Engines konfiguriert, aber auch mit dem *Codec* kommuniziert[10]. Jedes *Register* besitzt eine Größe von entweder 1 Byte, 2 Bytes oder 4 Bytes und ein festes Offset zur MMIO-Basisadresse, sodass die Adressen aller *Register* aus der MMIO-Basisadresse abgeleitet werden können. Die einzelnen Bits innerhalb eines *Registers* haben jeweils eine durch die Spezifikation vorgegebene Bedeutung und manche Bits dürfen vom Treiber nur gelesen und andere von ihm sowohl gelesen als auch verändert werden.
Beispielsweise ist das *Global Capabilities Register* 2 Bytes groß und besitzt das Offset 0x0[11] zur MMIO-Basisadresse. Das gesamte *Register* darf nur gelesen werden und teilt dem Treiber unter anderem mit, wie viele Output-DMA-Engines (Bits 15 bis 12), Input-DMA-Engines (Bits 11 bis 8) und bidirektionale DMA-Engines (Bits 7 bis 3) im *Controller* existieren.

[8] Eine DMA-Engine ist entweder eine Input- oder eine Output- oder eine bidirektionale DMA-Engine.
[9] Eine IHDA-Sounkarte benutzt immer MMIO und nie Port-Mapped I/O (PMIO).
[10] Die beiden vom *Controller* implementierten Schnittstellen zur Kommunikation zwischen Treiber und *Codec* werden später in 2.6.1.2 vorgestellt.
[11] Offset 0x0 bedeutet, dass sich dieses *Register* unmittelbar an der MMIO-Basisadresse befindet.

3.3.2 Offset 00h: GCAP – Global Capabilities

Length: 2 bytes

Table 3. Global Capabilities

Bit	Type	Description
15:12	RO	**Number of Output Streams Supported (OSS):** A value of 0000b indicates that there are no Output Streams supported. A value of maximum 15 output streams are supported. 0000b: No output streams supported 0001b: 1 output stream supported ... 1111b: 15 output streams supported
11:8	RO	**Number of Input Streams Supported (ISS):** A value of 0000b indicates that there are no Input Streams supported. A maximum of 15 input streams are supported. 0000b: No input streams supported 0001b: 1 input stream supported ... 1111b: 15 input streams supported
7:3	RO	**Number of Bidirectional Streams Supported (BSS):** A value of 00000b indicates that there are no Bidirectional Streams supported. A maximum of 30 bidirectional streams are supported. 00000b: No bidirectional streams supported 00001b: 1 bidirectional stream supported ... 11110b: 30 bidirectional streams supported
2:1	RO	**Number of Serial Data Out Signals (NSDO):** A 0 indicates that one SDO line is supported; a 1 indicates that two SDO lines are supported. Software can enable the use of striping by setting the appropriate bit in the Stream Buffer Descriptor. 00: 1 SDO 01: 2 SDOs 10: 4 SDOs 11: *Reserved*
0	RO	**64 Bit Address Supported (64OK):** A 1 indicates that 64 bit addressing is supported by the controller for BDL addresses, data buffer addresses, and command buffer addresses. A 0 indicates that only 32-bit addressing is available.

There are a maximum of 30 Streams that can be supported, of which 15 may be configured as output and 15 may be configured as input streams at any one point in time.

Abbildung 2.5: *Global Capabilities Register* des *Controllers*[29]

Die Anzahl der *Registers* eines *Controllers* hängt von der Anzahl der in die Soundkarte verbauten DMA-Engines ab, da jede DMA-Engine neun individuelle Status- und Kontrollregister besitzt, welche *Stream Descriptor Registers*[12] genannt werden. Von den individuellen Anzahlen der Input-, Output- und bidirektionalen DMA-Engines hängt zudem ab, an welchem Offset sich diese *Stream Descriptor Registers* für eine bestimmte Engine befinden. Um alle *Registers* korrekt lokalisieren zu können, muss der Treiber also zunächst die Anzahlen der drei Arten von Engines aus dem *Global Capacities Register* ermitteln.

[12] Was ein Stream ist, wird in 2.6.1.3 erläutert.

Link: Der *Link* bildet die physische Verbindung zwischen *Controller* und *Codec* und überträgt serialisierte Daten mit einer festen Übertragunsrate von 48 kHz in beide Richtungen. Dies bedeutet, dass 48000 Mal pro Sekunde bzw. alle 20.83 μs jeweils ein Datenpaket vom *Controller* zum *Codec* und eines vom *Codec* zum *Controller* gesendet wird. Diese Datenpakete werden *Frames* genannt und enthalten sowohl Audiodaten in Form einzelner Samples als auch Botschaften zur Kommunikation zwischen Treiber und *Codec*, welche in Abschnitt 2.6.1.2 vorgestellt werden.

Codec: Als *Codec* wird jegliches Gerät bezeichnet, welches über den *Link* mit dem *Controller* verbunden ist.[30] Diese Definition ist so allgemein gehalten, da IHDA neben den zentralen *Audio Codecs* zusätzlich *Modem Codecs* und sogar *Vendor Defined Codecs*, also vollständig von einem Dritthersteller definierte *Codecs*, unterstützt.[31] Auf sowohl *Modem Codecs* als auch *Vendor Defined Codecs* wird in der Spezifikation allerdings nicht weiter eingegangen, sodass auch in dieser Arbeit davon ausgegangen wird, dass jeder *Codec* ein *Audio Codec* ist. In dieser Arbeit werden außerdem keine *Codecs* betrachtet, welche ausschließlich digitale Schnittstellen zur Außenwelt wie HDMI, Displayport oder S/PDIF bereitstellen[13] und es wird davon ausgegangen, dass ein *Codec* immer mindestens einen DAC und einen ADC besitzt (vgl. 2.5), sodass mindestens ein analoger Lautsprecher und ein analoges Mikrofon an den *Codec* angeschlossen werden können.

Die modulare Architektur[32] eines *Codecs* stellt verschiedene parametrisierte Module zur Verfügung, aus denen unter Berücksichtigung der ebenfalls durch die Spezifikation vorgegebenen Regeln zur Verknüpfung dieser Module miteinander ein *Codec* konstruiert werden kann. Die einzelnen Module eines *Codecs* werden *Nodes* genannt und in drei Arten unterteilt: *Root Node*, *Function Group Node* und *Widget Node*.[33]
Jede *Node* besitzt eine eindeutige Adresse, welche sich aus der Adresse des *Codecs*, in den sie verbaut ist, und einer innerhalb dieses *Codecs* eindeutigen *Node ID* zusammensetzt. Außerdem besitzt jede *Node* eine Menge an read-only *Parameters*, über die statische und dynamische Informationen über die *Node* eingeholt werden können [34], sowie eine Menge an read-write *Controls*, durch welche die *Node* konfiguriert werden kann[35]. Sowohl das Abfragen eines *Parameters* als auch das Abfragen oder Ändern einer *Control* geschieht über einen der beiden Mechanismen zur Kommunikation zwischen Treiber und *Codec*, welche beide vom *Controller* implementiert und in 2.6.1.2 vorgestellt werden.

[13] Solche *Codecs* finden sich beispielsweise auf IHDA-Geräten, welche in Grafikkarten eingebaut sind, damit die Grafikkarte Audio über HDMI oder Displayport wiedergeben kann. Sie funktionieren ähnlich wie die in dieser Arbeit behandelten *Codecs*, benötigen allerdings keine DACs oder ADCs zur Signalverarbeitung.

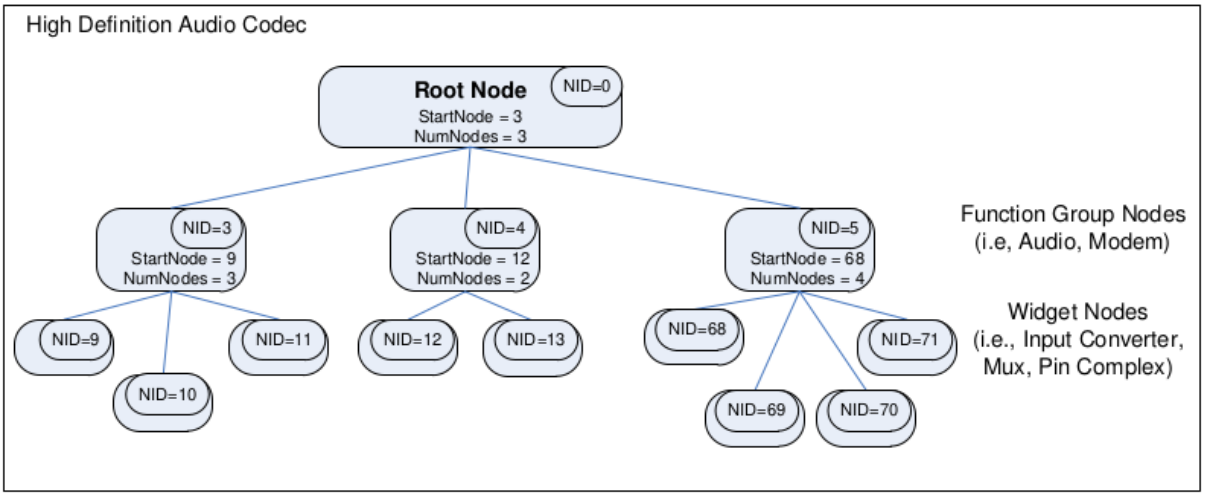


Abbildung 2.6: Adressierungsschema innerhalb eines *Codecs*[36]

Die *Nodes* innerhalb eines *Codecs* sind in einer Baumstruktur organisiert.[32] Jeder *Codec* besitzt genau eine *Root Node* mit der *Node ID* 0 und mindestens[14] eine *Function Group Node*. Die *Root Node* besitzt keine *Controls* und ihre *Parameters* stellen neben Informationen über den Hersteller des *Codecs* und die Revision der IHDA-Spezifikation, nach welcher der *Codec* gebaut wurde, auch die *Node ID* der *Function Group Node* bereit. Diese *Function Group Node* beschreibt eine *Function Group* genannte Menge von elektrischen Bauteilen, welche innerhalb des Codecs direkt oder über andere Bauteile physisch miteinander verbunden sind. Jedes einzelne dieser Bauteile besitzt eine klar abgegrenzte Funktion und wird durch genau eine *Widget Node* (kurz: *Widget*) repräsentiert. Die *Function Group Node* besitzt einen *Parameter*, über den die *Node IDs* der zu ihr gehörenden *Widgets* ermittelt werden können. Beispiele für solche *Widgets* sind ein *Audio Output Converter Widget*[15] (*AOCW*), welches ein digitales Audiosignal in ein analoges umwandelt, ein *Mixer Widget* (*MW*), welches zwei analoge Audiosignale innerhalb eines Codecs zu einem einzigen aufsummiert, oder ein *Line Out Pin Widget* (*LOPW*), welches fest mit einer einzigen analogen Ausgangsbuchse verkabelt ist und ein analoges Audiosignal über diese Buchse an ein Peripheriegerät, wie beispielsweise einen Lautsprecher, übertragen kann.
Um ein über den *Link* erhaltenes digitales Signal über solch einen analogen Ausgang des *Codecs* wiedergeben zu können, muss dieses Signal auf dem *Codec* zunächst in ein analoges Signal konvertiert und dann zu einer Ausgangsbuchse weitergeleitet werden. Mindestens ein *AOCW* und ein *LOPW* werden also für die Audiowiedergabe benötigt.[16] Es gibt jedoch IHDA-Soundkarten, bei denen ein *AOCW* nicht unmittelbar mit einem *LOPW*

[14] Sowohl der einzige Codec auf der virtualisierten Soundkarte in QEMU als auch der einzige Codec auf der integrierten Soundkarte des für die Entwicklung genutzten Testrechners besaßen jeweils nur eine einzige *Function Group Node*. Aus diesem Grund wird im Rest dieser Arbeit meist nur von der *Function Group Node* und nicht von den *Function Group Nodes* gesprochen.

[15] Dies ist innerhalb der IHDA-Spezifikation die Bezeichnung für einen DAC.

[16] Im Fall der Audioaufzeichnung wird mindestens ein *Mic In Pin Widget* und ein *Audio Input Converter Widget* benötigt. Dieser Fall verläuft analog zum beschriebenen Fall der Audiowiedergabe, allerdings wird das Signal dort in umgekehrter Richtung vom *Codec* zum *Controller* übertragen. Dieser Fall wird nicht weiter betrachtet.

verbunden ist, weil andere *Widgets*, wie zum Beispiel ein *MW*, zwischengeschaltet sind.[17]

2.6.1.2 Kommunikation zwischen Treiber und Codec

Nachdem im vorherigen Abschnitt erklärt wurde, dass jede *Node* innerhalb des *Codecs* eine eindeutige *Node ID* sowie *Parameters* und *Controls* besitzt, soll nun erläutert werden, wie der Treiber die *Paramaters* und *Controls* einer *Node* abfragen bzw. ihre *Controls* ändern kann.
Die IHDA-Spezifikation nennt zwei Mechanismen zur Kommunikation zwischen Treiber und *Codec*, welche beide vom *Controller* implementiert und im Folgenden vorgestellt werden. Beide Mechanismen haben gemeinsam, dass der Treiber sie dafür nutzen kann, ein *Command* über den an den *Controller* angeschlossenen *Link* an eine *Node* im *Codec* zu senden, woraufhin diese *Node* ebenfalls über den *Link* mit einer *Response* antwortet. Ein *Command* setzt sich aus der vollständigen Adresse der *Node*, also Adresse des *Codecs* und *Node ID*, sowie der sogenannten *Verb ID* des *Commands*, welche dieses *Command* eindeutig identifiziert, und einem *Payload*, dessen Format wiederum von der *Verb ID* abhängig ist, zusammen.[37] In jedem *Frame* des *Links* kann maximal ein *Command* vom *Controller* zum *Codec* und maximal eine *Response* vom *Codec* zum *Controller* übertragen werden.[38] Sowohl jedes *Command* als auch jede *Response* besitzt eine Größe von 4 Bytes, jedoch ist die Bedeutung der Bits bei jedem *Command* bzw. jeder *Response* unterschiedlich. Allerdings ist das Format einer Nachricht, mit Ausnahme der Bits zur Adressierung des *Codecs*, für den *Controller* vollkommen opak, sodass aus dessen Sicht jedes *Command* und jede *Response* das gleiche Format eines 4-Bytes-Pakets besitzt. Die einzelnen Bits eines *Commands* bzw. einer *Response* haben somit, bis auf die Bits zur Adressierung des *Codecs*, ausschließlich für Treiber und *Codec* eine Bedeutung.[39]
Es gibt zwei Arten von *Responses*. Eine *Response*, welche als Reaktion auf ein vom Treiber versendetes *Command* erhalten wurde, heißt *Solicited Response*. Im Gegensatz dazu wird eine *Unsolicited Responses* von einem *Codec* unabhängig von einer Anfrage durch den Treiber gesendet und signalisiert dem Treiber, dass der *Codec* dessen Aufmerksamkeit braucht.[39]

Die IHDA-Spezifikation stellt zwei Schnittstellen für die Kommunikation zwischen Treiber und *Codecs* zur Verfügung, welche nicht gleichzeitig genutzt werden sollten:

Command Outbound Ring Buffer und Response Inbound Ring Buffer

Die erste Art der Kommunikation zwischen Treiber und *Codec* verläuft über zwei Ringbuffer, den *Command Outbound Ring Buffer* (*CORB*)[40] und den *Response Inbound Ring Buffer* (*RIRB*)[41], welche jeweils von einer eigenen DMA-Engine im *Controller* betrieben werden. Nachdem der Treiber die beiden Ringbuffer initialisiert und gestartet hat, kann er *Commands* im *CORB* platzieren, woraufin der *Controller* diese *Commands* sequenziell[18] über den *Link* an den *Codec* sendet und die erhaltenen *Responses* in den *RIRB* schreibt. In der Regel bieten beide Ringbuffer Platz für 256 Einträge, also für 256 *Commands* beim

[17] Dies war zum Beispiel beim *Codec* der in den für die Entwicklung genutzten Testrechner verbauten integrierten Soundkarte der Fall. Ein Verknüpfungsgraph der *Widgets* dieses konkreten *Codecs* befindet sich in 6.1.

[18] ein *Command* pro *Frame* (vgl. 2.6.1.1)

CORB und 256 *Responses* beim *RIRB*. Jeder Eintrag im *CORB* entspricht genau einem zu versendenden *Command* mit der festen Größe von 4 Bytes und jeder Eintrag im *RIRB* entspricht genau einer *Response* mit der festen Größe von 4 Bytes plus einem ebenfalls 4 Bytes großen Header, welcher die Adresse des antwortenden *Codecs* sowie die Information, ob es sich um eine *Solicited Response* oder *Unsolicited Response* handelt, enthält. Der Header enthält jedoch keine Information darüber, welche individuelle *Node* innerhalb des *Codecs* diese *Response* abgesendet hat und auch keine Information darüber, auf welches *Command* geantwortet wurde. Die Zuordnung einer *Response* zu dem dazugehörenden *Command*, ohne welche das erhaltene 4-Bytes-Paket nicht interpretiert werden kann, ist Aufgabe des Treibers. Damit die Zuordnung gelingen kann, wird die zu einem im *CORB* platzierten *Command* gehörende *Response* vom Controller immer an den gleichen Index im *RIRB* geschrieben, an dem sich das *Command* im *CORB* befindet.

Bevor der Treiber mit der Initialsierung von *CORB* und *RIRB* beginnt, sollte er zunächst sicherstellen, dass ihre DMA-Engines ausgeschaltet sind, indem er sowohl das *CORBRUN Bit* im *CORB Control Register* des *Controllers* als auch das *RIRBRUN-Bit* im *RIRB Control Register* löscht. Als nächstes alloziert der Treiber für jeden der beiden Ringbuffer einen eigenen DMA-Bereich und teilt dem Controller die Startadressen dieser Bereiche über entsprechende *Register* mit. Anschließend setzt er die insgesamt drei benötigten Zeiger, welche für den Betrieb von *CORB* und *RIRB* benötigt werden und sich ebenfalls jeweils in einem eigenen *Register* des *Controllers* befinden, zurück. Diese Zeiger sind der *CORB Write Pointer* (*CWP*), der *CORB Read Pointer* (*CRP*) und der *RIRB Write Pointer* (*RWP*). Nach diesen Vorbereitungen können die beiden Ringbuffer in Betrieb genommen werden, indem ihre DMA-Engines durch das Setzen des *CORBRUN Bits* und des *RIRBRUN Bits* wieder gestartet werden.

Um die Funktionsweise der Ringbuffer besser nachvollziehen zu können, sei zunächst angenommen, dass der *Codec* keine *Unsolicited Responses* eigenständig im *RIRB* platziert. Gibt es unter dieser Annahme keine vom *Controller* noch nicht versendeten *Commands* im *CORB*, enthalten *CWP*, *CRP* und *RWP* denselben Wert. Der Treiber versendet ein *Command* dann wie folgt:

1. Zunächst überprüft der Treiber, auf welchen Eintrag der *RWP* zeigt, und schreibt das zu versendende *Command* an den nächst höheren Index in den *CORB*.

2. Anschließend inkrementiert der Treiber den *CWP* um 1 und signalisiert dem *Controller* damit, dass ein neues *Command* versendet werden soll.

3. War das Versenden erfolgreich, teilt der *Controller* dies dem Treiber mit, indem er den *CRP* um 1 inkrementiert.

4. War auch das Schreiben der *Response* in den *RIRB* erfolgreich, erhöht der *Controller* ebenfalls den *RWP* um 1. Somit besitzen alle drei Zeiger wieder denselben Wert und der Prozess kann wiederholt werden.

Der Treiber kann auch mehrere *Commands* auf einmal im *CORB* platzieren und den *CWP* um die entsprechende Anzahl der *Commands* inkrementieren, woraufhin der *Controller* so lange *Commands* versendet, bis der *CRP* den *CWP* eingeholt hat. Da der *Controller* jedoch pro *Frame* des *Links* lediglich ein einzelnes *Command* übertragen kann, kann

das gleichzeitige Versenden sehr vieler *Commands* durch den Treiber dazu führen, dass der *CWP* den *CRP* überholt und somit den *CORB* zum Überlaufen bringt. Der Treiber muss darauf achten, dass dies nie passiert und gegebenenfalls mit dem Schreiben neuer *Commands* in den *CORB* warten, bis der *Controller* genügend angestaute *Commands* verarbeitet hat und somit wieder Platz im *CORB* frei geworden ist. Für das rechtzeitige Auslesen eines Eintrags im *RIRB*, bevor dieser Eintrag 256 *Responses* später überschrieben wird, ist ebenfalls der Treiber verantwortlich. Zu diesem Zweck kann der Treiber den *RIRB* so konfigurieren, dass er nach einer vom Treiber festgelegten Anzahl von erhaltenen *Responses* einen Hardware-Interrupt auslöst.

Wenn es dem *Codec* nun doch erlaubt ist *Unsolicited Responses* zu verschicken, werden diese *Unsolicited Responses* vom *Controller* neben den vom Treiber angefragten *Solicited Responses* ebenfalls im *RIRB* platziert, korrespondieren allerdings zu keinem Eintrag im *CORB*. Dies führt dazu, dass der *RWP* einen höheren Wert haben kann als *CWP* und *CRP*, obwohl keine zu versendenden *Commands* im *CORB* existieren. Deshalb muss der Treiber vor dem Versenden neuer *Commands* stets überprüfen, ob die Werte von *RWP* und *CWP* identisch sind. Ist dies nicht der Fall, liegen neue *Unsolicited Responses* im *RIRB*, welche vor dem Senden neuer *Commands* verarbeitet werden sollten.

Immediate Command Input and Output Registers

Für die zweite Art der Kommunikation zwischen Treiber und *Codec* stellt der *Controller* ein aus drei *Registers* bestehendes Programmed-I/O-Interface[42] zur Verfügung, welches vom Treiber wie folgt bedient wird:

1. Zunächst platziert der Treiber das zu sendende *Command* im *Immediate Command Output Interface Register* (*ICOI Register*) und setzt anschließend das *Immediate Command Busy Bit* (*ICB Bit*) im *Immediate Command Interface Status Register* (*ICIS Register*). Dadurch signalisiert er dem Controller, dass ein *Command* für den Versand über den *Link* bereit ist.

2. Sobald der *Controller* die *Response* vom *Codec* erhalten hat, platziert er sie im *Immediate Command Input Interface Register* (*ICII Register*) und setzt das *Immediate Result Valid Bit* (*IRV Bit*) im *ICIS Register*. Dadurch meldet er dem Treiber zurück, dass die *Response* nun verfügbar ist.

3. Nachdem der Treiber die *Response* verarbeitet hat, setzt er das *ICB Bit* zurück und löscht auch das *IRV Bit*, indem er es erneut setzt[19]. Daraufhin befinden sich alle drei *Registers* wieder in ihren initialen Zuständen und der Prozess kann wiederholt werden.

Da *Commands* bei dieser Art der Kommunikation nur einzeln versendet werden können, ist die Kommunikation über die *Immediate Command Input and Output Registers ICIO Registers* einerseits langsamer als die Kommunikation über *CORB* und *RIRB* und benötigt andererseits im direkten Vergleich auch mehr Aufmerksamkeit der CPU. Deshalb sind *CORB* und *RIRB* generell zu bevorzugen, insbesondere wenn während des Betriebs der Soundkarte viele *Commands* verschickt werden müssen. Die Spezifikation bezeichnet

[19] „write-1-to-clear“

die *ICIO Registers* sogar als optional, sodass ein Hersteller gar nicht erst dazu verpflichtet ist, diese zu implementieren. Da deren Bedienung durch den Treiber in Form von simplen Registerzugriffen jedoch deutlich einfacher ist als die Benutzung von *CORB* und *RIRB*, bieten sie, falls vorhanden[20], eine unkomplizierte Möglichkeit während der Entwicklung des Treibers mit dem *Codec* zu experimentieren, ohne sich um die zahlreichen Tücken bei der Implementierung zweier Ringbuffer Gedanken machen zu müssen.

2.6.1.3 Streams und Channels

Bisher wurde in dieser Arbeit bereits mehrfach über Audiodaten oder digitale Audiosignale gesprochen, welche über den *Link* übertragen werden. Im letzten Teil des Kapitels über die Architektur eines IHDA-Geräts soll diese Formulierung nun präzisiert werden, indem das Konzept von *Streams* und *Channels* vorgestellt wird, durch welches die Daten für die Übertragung über den *Link* organisiert werden.[43]
Ein *Stream* ist eine virtuelle Verbindung zwischen mindestens zwei Audiobuffern im Systemspeicher und einem oder mehreren *Codecs*. Diese Verbindung wird durch eine einzelne DMA-Engine im *Controller* betrieben. (vgl. Abb. 2.7)[21]

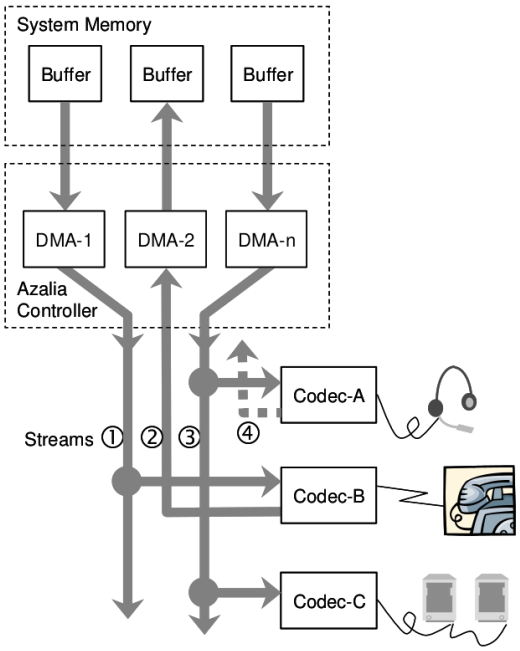


Abbildung 2.7: Streams[45]

Ein *Stream* ist entweder ein *Output Stream* oder ein *Input Stream* und enthält mindestens einen und bis zu sechzehn *Channels*, welche jeweils ein einziges zeitdiskretes Signal (vgl. 2.5) repräsentieren. Jeder *Stream* besitzt eine eindeutige *Stream ID* und ein *Stream Format*, welches die für den *Stream* genutzte Samplerate und Bittiefe sowie die Anzahl der im *Stream* enthaltenen *Channels* bestimmt[46]. Dieses *Stream Format* gibt vor, wie

[20] Sowohl der *Controller* der virtualisierten Soundkarte in QEMU als auch der *Controller* der integrierten Soundkarte des für die Entwicklung genutzten Testrechners implementierte die *ICIO Registers*.

[21] Der in der Abbildung enthaltene Begriff „Azalia“ war der interne Codename von HD Audio bei Intel.[44] Der „Azalia Controller“ ist also schlicht der *Controller*.

die einzelnen Bytes in den Audiobuffern des *Streams* von der DMA-Engine im *Controller* interpretiert werden und in Form von einzelnen Samples im Fall eines *Output Streams* auf den *Link* geschickt oder im Fall eines *Input Streams* vom *Link* empfangen werden sollen.[47]
Jeder *Stream* besitzt genau eine *Buffer Descriptor List* (*BDL*) mit mindestens zwei Einträgen und jeder einzelne *Buffer Descriptor List Entry* beschreibt genau einen für den *Stream* genutzten Audiobuffer durch Adresse und Länge des Buffers sowie die Information, ob dieser Buffer einen Interrupt auslösen soll, nachdem er von der DMA-Engine komplett verarbeitet wurde (*Interrupt on Completion*). Die Menge aller durch die *BDL* eines *Streams* beschriebenen Audiobuffer wird *Cyclic Buffer* genannt. Sowohl die *BDL* als auch die Audiobuffer befinden sich jeweils in eigenen DMA-Bereichen, welche vom Treiber alloziert und deren Startadressen dem *Controller* über die *Stream Descriptor Registers* der zuständigen DMA-Engine (vgl. 2.6.1.1) mitgeteilt werden müssen.
Wird die DMA-Engine eines *Streams* gestartet, durchläuft sie die einzelnen Buffer zyklisch, beginnend beim ersten Eintrag in der *BDL*, wobei die einzelnen Bytes in den Buffern gemäß dem *Stream Format* des *Streams* als Samples interpretiert werden.[48] Nachdem die DMA-Engine den letzten Audiobuffer vollständig durchlaufen hat, beginnt sie ohne Pause wieder beim ersten und hört erst mit der Übertragung von Samples über den *Link* auf, wenn sie wieder angehalten wird[49]. Der Treiber ist für die Daten in den Audiobuffern verantwortlich und muss dafür Sorge tragen, dass diese Daten rechtzeitig aktualisiert werden.

Als Beispiel sei ein *Output Stream* und dessen *BDL* mit lediglich zwei Einträgen gegeben. Die beiden durch diese Einträge beschriebenen Audiobuffer enthalten jeweils so viele Samples, dass das Abspielen dieser Samples bei der durch das *Stream Format* des *Streams* vorgegebenen Samplerate eine Sekunde dauert. In der ersten Sekunde nach ihrem Start liest die DMA-Engine den ersten Audiobuffer des *Streams* aus und springt zu Beginn der zweiten Sekunde in den zweiten Buffer. Der Treiber hat nun genau eine Sekunde Zeit, die Daten im ersten Buffer zu aktualisieren, da die DMA-Engine zu Beginn der dritten Sekunde wieder in den ersten Buffer springt. Die *Interrupts on Completion* helfen dem Treiber dabei festzustellen, wann die Daten in einem Buffer aktualisiert werden müssen. Werden die Daten nie durch den Treiber aktualisiert, ist bis zum Stopp der DMA-Engine ein zwei-sekündiger Audio-Loop zu hören.
Im Fall eines *Input Streams* verläuft das Beispiel analog mit dem Unterschied, dass von der DMA-Engine in die Audiobuffer geschrieben anstatt aus ihnen gelesen wird. Daten, welche nicht aus einem Buffer ausgelesen werden, bevor dieser Buffer wieder an der Reihe ist, werden blind überschrieben und gehen somit verloren.

Damit der Treiber überwachen kann, ob die DMA-Engines aller aktiven *Streams* ihre individuellen *Cyclic Buffer* korrekt durchlaufen, kann optional ein weiterer DMA-Bereich für den sogenannten *DMA Position Buffer* alloziert werden.[50] Wird dieser *DMA Position Buffer* aktiviert, schreibt der *Controller* für jeden *Stream* das Offset der Adresse innerhalb des *Cyclic Buffers*, auf welche die DMA-Engine des *Streams* zuletzt zugeriffen hat, in den *DMA Position Buffer* und aktualisiert dieses Offset nach jedem *Frame* des *Links*.

2.6.2 Programmiermodell

Nachdem in 2.6.1 die Architektur einer IHDA-Soundkarte beschrieben wurde, wird in diesem Kapitel erläutert, welche Aufgaben der Treiber erledigen muss, bevor ein *Output Stream* über einen analogen Ausgang der Soundkarte wiedergegeben werden kann. Jedes der folgenden Unterkapitel beschreibt eine dieser Aufgaben und die Kapitelreihenfolge suggeriert eine mögliche korrekte Reihenfolge der Ausführung. Während einige Schritte miteinander vertauscht werden können und somit andere Ausführungsreihenfolgen möglich sind, muss beispielsweise das Abbilden der *Registers* (vgl. 2.6.2.1) zwingend vor allen darauf folgenden Schritten erfolgen.

2.6.2.1 Initialisierung des PCI-Interfaces

Der über PCI mit dem Systemspeicher verbundene *Controller* besitzt wie jedes PCI-Gerät Register in seinem individuellen PCI Configuration Space[22], welche Informationen über das Gerät enthalten und über die das Gerät konfiguriert werden kann. Bevor der Treiber mit dem *Controller* kommunizieren kann, müssen folgende Schritte unternommen werden:

Finden des IHDA-Geräts auf dem PCI-Bus Zunächst muss der Treiber das IHDA-Gerät auf dem PCI-Bus suchen, wofür er die vom Betriebssystem bereitgestellten Ressourcen benutzt. Solch ein Gerät besitzt normalerweise die PCI Class 4 (Multimedia Device) und die PCI Subclass 3 (Intel HD Audio Device) und kann über diese Informationen gefunden werden.[51]

Konfigurierung des PCI Configuration Space In 2.6.1.1 wurde erklärt, dass dem *Controller* erlaubt werden muss, als Bus Master direkt in den Speicher zu schreiben, wofür das Bus Master Bit im Command Register des PCI Configuration Space gesetzt werden muss (vgl. 2.4). Außerdem muss im selben Register das Memory Space Bit gesetzt werden, damit der *Controller* auf Speicherzugriffe antworten kann.[52]

Verbinden der Interrupt Line In der Regel teilt der *Controller* dem Betriebssystem bzw. dem Treiber über Hardware-Interrupts mit, wann neue Daten in einen Audiobuffer geschrieben oder aus einem Audiobuffer gelesen werden müssen, und auch für andere Zwecke nutzt der *Controller* Interrupts. Um Interrupts erzeugen zu können, muss der *Controller* mit dem Interrupt-Controller[23] des Betriebssystems verbunden werden, indem die vom *Controller* genutzte Interrupt Line aus dem entsprechenden Register seines PCI Configuration Space gelesen und dem Interrupt-Controller des Betriebssystems mitgeteilt wird.[53]

Abbilden der Register des Controllers Die *Registers*[24] des *Controllers* werden während der Enumerierung und Initialisierung des PCI-Busses an eine feste Adresse im physischen Adressraum abgebildet. Diese MMIO-Basisadresse und die Größe des für die *Registers* allozierten Speicherblocks können aus dem Bar0 Register im PCI Configuration

[22] Diese in der PCI-Spezifikation definierten Register sind nicht zu verwechseln mit den in 2.6.1.1 vorgestellten und in der IHDA-Spezifikation definierten *Registers* des *Controllers*.

[23] Im Falle des D3OS ist dies der APIC.

[24] Nun sind tatsächlich wieder die *Registers* aus 2.6.1.1 gemeint.

Space gelesen werden.[54] Der Treiber muss einen gleich großen Speicherblock im virtuellen Kernel-Adressraums allozieren und diesen auf dieselbe physische MMIO-Basisadresse abbilden. Durch dieses doppelte Mapping bedeutet ein Lesen oder Schreiben in den virtuellen Kernel-Adressraum, in dem der Treiber operiert, immer auch ein unmittelbares Schreiben in die *Register* auf der Soundkarte und umgekehrt. Erst nachdem diese Verbindung hergestellt wurde, kann der Treiber mit dem *Controller* kommunizieren.

2.6.2.2 Start und Konfigurierung des Controllers

Nach dem Start des Betriebssystems befinden sich alle *Registers* des *Controllers* in ihren Power-On-Default-Zuständen und der *Link* ist inaktiv. Das einzige Bit, welches zu diesem Zeitpunkt Schreibzugriffe akzeptiert, ist das *Controller Reset Bit* (*CRST Bit*) im *Global Control Register*. Schreibzugriffe in andere *Register* haben keinen Effekt. Unmittelbar nach dem Systemstart ist das *CRST Bit* nicht gesetzt und signalisiert dadurch, dass sich der *Controller* im Reset-Zustand befindet. Der Treiber startet den *Controller* durch das Setzen des *CRST Bits*, woraufhin der *Link* zurückgesetzt und eine Selbstinitialisierung des *Codecs* ausgelöst wird.[55] Da dieser Vorgang etwas Zeit benötigt, sollte der Treiber ausreichend lange[25] warten, bevor er mit der Initialisierung fortfährt.[56]
Einige Features einer IHDA-Soundkarte können über einzelne Bits in entsprechenden *Registers* ein- und ausgeschaltet werden. Beispielsweise kann dem Codec grundsätzlich verboten werden *Unsolicited Responses* zu verschicken[57] oder es können Interrupts entweder für einzelne DMA-Engines oder den gesamten *Controller* deaktiviert werden[58]. Diese Einstellungen können zwar jederzeit geändert werden, jedoch bietet sich eine initiale Konfigurierung zu diesem Zeitpunkt an.
Falls für die Kommunikation zwischen Treiber und *Codecs* die beiden Ringbuffer *CORB* und *RIRB* genutzt werden sollen, müssen diese vor der im folgenden Abschnitt vorgestellten Ermittlung der Topologie des *Codecs* wie in 2.6.1.2 beschrieben initialisiert werden. Diese ist nicht notwendig, falls stattdessen die *ICIO Registers* (vgl. 2.6.1.2) genutzt werden. Falls der Treiber die DMA-Engines während des Betriebs über den in 2.6.1.3 vorgestellten DMA Position Buffer überwachen soll, muss ein DMA-Bereich für diesen Buffer alloziert und dessen Startadresse dem Controller über entsprechende Register mitgeteilt werden.

2.6.2.3 Ermittlung der Topologie des Codecs

Da sowohl Anzahl, *Node IDs*, und Typen der in einen *Codec* verbauten *Nodes* als auch deren Verknüpfungen untereinander von Gerät zu Gerät unterschiedlich sind, ist die Topologie des *Codecs* dem Treiber nach dem Start des *Controllers* vollkommen unbekannt.[59] Darum muss der Treiber, wie bei der Vorstellung des *Codecs* in 2.6.1.1 bereits angedeutet, über einen der beiden vom *Controller* implementierten Mechanismen zur Kommunikation mit dem *Codec* zunächst die *Root Node*, welche die einzige bekannte *Node ID* 0 besitzt, nach den Adressen aller *Function Group Nodes* und diese *Function Group Nodes* wiederum nach den Adressen aller *Widgets* befragen.[26] Danach weiß der Treiber zwar bereits, wie viele *Widgets* in einer *Function Group* existieren und wo diese sich befinden, kennt aber weder deren individuelle Typen noch deren Verknüpfungen untereinander. Der Typ eines *Widgets* kann über einen *Parameter* namens *Audio Widget Capabilities*[60] eingeholt

[25] mindestens 25 *Frames* bzw. 521 μs

[26] Dafür muss jeweils ein *Parameter* namens *Subordinate Node Count* angefragt werden.

werden. Einige *Widgets*, wie beispielsweise *DOCW* und *MW*, besitzen außerdem eine über einen *Parameter* namens *Connection List*[61] abrufbare Liste, welche Auskunft darüber gibt, mit welchen anderen *Widgets* ein *Widget* verbunden ist.[27] Jeder Eintrag in dieser Liste repräsentiert eine physische Verbindung zwischen dem *Widget* und genau einem anderen *Widget*.[28]
Nachdem der Treiber all diese *Parameters* von den verschiedenen *Widgets* eingeholt hat, kennt er Adressen, Typen und Verknüpfungen aller *Nodes* und besitzt somit ein vollständiges Bild von der Topologie des *Codecs*.

2.6.2.4 Erstellung eines Streams

Das Erstellen und Verwalten von *Streams* liegt vollständig in der Verantwortung des Treibers. Zunächst muss der Treiber ein *Stream Format* bestimmen und überprüfen, ob *Controller* und *Codec* dieses *Stream Format* unterstützen. Anschließend müssen DMA-Bereiche für eine *BDL* sowie die durch diese Liste beschriebenen Audiobuffer alloziert werden. Der Treiber reserviert dann eine DMA-Engine und schreibt *Stream Format*, Adresse der *BDL*, Länge des *Cyclic Buffers* in Bytes, die genutzte Interrupt Policy sowie eine selbst gewählte *Stream ID* an entsprechende Stellen innerhalb der zur genutzten DMA-Engine gehörenden *Stream Descriptor Registers*. Der *Controller* ist nun bereit, den *Stream* zu starten und mit dessen Übertragung über den *Link* zu beginnen.[48] Allerdings muss zuvor ebenfalls der *Codec* konfiguriert werden, damit der aktive *Stream* auch verarbeitet wird.

2.6.2.5 Konfigurierung des Codecs für die Audiowiedergabe

Die Konfigurierung des *Codecs* erfolgt wie bereits die Ermittlung der Topologie des *Codecs* über das Senden von *Commands* an die *Nodes* des *Codecs*. Als Beispiel sei im Folgenden der Fall betrachtet, dass ein einziger *Output Stream* über einen analogen Ausgang des *Codecs* wiedergegeben werden soll.
In 2.6.1.1 wurde erklärt, dass für die Audiowiedergabe über einen *Codec* mindestens ein *DOCW* und ein *LOPW* benötigt werden. Zunächst muss der Treiber also einen geeigneten Pfad zwischen einem *DOCW* und demjenigen *LOPW*, über dessen Ausgangsbuchse das Signal wiedergegeben werden soll, innerhalb der mittlerweile bekannten Topologie des *Codecs* finden. Anschließend muss der Treiber jedes *Widget* auf dem Pfad über die *Controls* des *Widgets* individuell konfigurieren, indem er den richtigen Nachfolger auf dem bestimmten Ausgabepfad aus der ebenfalls bereits erwähnten *Connection List* des *Widgets* auswählt[29] und alle eventuell vorhandenen Audio-Verstärker auf dem Ausgabeweg[30] einschaltet sowie deren Lautstärkepegel einstellt. Dem *DOCW* muss er außerdem *Stream Format* sowie *Stream ID* des in 2.6.2.4 erstellten *Streams* mitteilen, sodass das *DOCW* mit der korrekten DMA-Engine verbunden wird.[62]
Nachdem alle *Nodes* auf dem Ausgabepfad konfiguriert wurden, ist der *Codec* bereit, die Daten des *Streams* vom *Link* zu empfangen und die Übertragung des *Streams* kann durch

[27] Ein *LOPW* besitzt keinen *Parameter* namens *Connection List*, da es lediglich mit einer einzigen Ausgangsbuchse und keinen anderen *Widgets* verbunden ist.

[28] Ein Anschauungsbeispiel findet sich in 6.2.

[29] Zu jedem Zeitpunkt kann maximal eine Verbindung aus der *Connection List* eines jeden *Widgets* aktiv sein.

[30] Sowohl *DOCW*, *MW* als auch *LOPW* können je nach Bauweise konfigurierbare Verstärker enthalten.

das Setzen des *RUN Bits* im entsprechenden *Stream Descriptor Register* der DMA-Engine gestartet werden. Solange das *RUN Bit* gesetzt ist, werden die Audiodaten im *Cyclic Buffer* des *Streams* wie in 2.6.1.3 beschrieben an den mit dem *LOPW* verbundenen analogen Ausgang des *Codecs* gesendet und können über einen an diesen Ausgang angeschlossenen Lautsprecher gehört werden.

2.6.3 Zusammenfassung – Intel HD Audio

Um Audio über einen analogen Ausgang einer IHDA-Soundkarte wiederzugeben, überträgt der *Controller* PCM-Daten aus dem Speicher über eine seiner DMA-Engines in Form eines *Streams* über den *Link* zum *Codec*, wo dieser *Stream* zunächst in ein analoges Signal umgewandelt und dann auf einem aus mindestens zwei *Widgets* bestehenden Ausgabepfad über eine analoge Ausgangsbuchse an ein Wiedergabegerät, wie beispielsweise einen Lautsprecher, weitergeleitet wird. Sowohl *Controller* als auch *Codec* müssen dafür vom Treiber initialisiert und entsprechend konfiguriert werden. Die Konfigurierung des *Controllers* wird vom Treiber unmittelbar über die vom physischen Adressraum in den virtuellen Kernel-Adressraum abgebildeten *Registers* durchgeführt und die Konfigurierung des *Codecs* geschieht über einen der beiden vorgestellten Mechanismen zur Kommunikation zwischen Treiber und *Codec*.
Insgesamt greift der *Controller* dabei zu fünf verschiedenen Zwecken auf DMA-Bereiche zu: den Zugriff auf seine in den physischen Adressraum abgebildeten *Registers*, den Betrieb von *CORB* und *RIRB*, den Betrieb des für die Überwachung der DMA-Engines durch den Treiber genutzten *DMA Position Buffers*, den Zugriff auf die *BDL* eines Streams sowie zuletzt für den Zugriff auf die eigentlichen Audiobuffer.

Kapitel 3

Implementierung

Nachdem im vorherigen Kapitel 2 die allgemeinen Grundlagen zur Entwicklung eines Intel HD Audio Soundkartentreibers erklärt wurden, beschäftigt sich dieses Kapitel mit der konkreten Implementierung des im Rahmen dieser Arbeit entwickelten Treibers[20]. Dafür wird zunächst in 3.1 ein Überblick über die Struktur des Treibers gegeben. Anschließend wird in 3.2 erläutert, wie der Treiber in das D3OS integriert wurde, bevor die Module des Treibers in 3.3 einzeln vorgestellt werden.

3.1 Struktur des Treibers

Bei der Implementierung des Treibers in Rust wurde ein objektorientierter Ansatz verfolgt, bei dem die in 2.6.1 vorgestellten Entitäten der IHDA-Architektur, wie zum Beispiel *Controller*, *Stream* oder *Codec*, in Structs bzw. Enums übersetzt wurden. Dabei wurde vermieden künstliche Abstraktionen einzuführen, zu denen sich keine Entsprechung innerhalb der Spezifikation finden lässt. Die in 2.6.2 beschriebene Funktionalität, wie beispielsweise die Ermittlung einer Topologie des *Codecs*, wird vornehmlich durch die auf diesen Structs bzw. Enums definierten Methoden bereitgestellt.
Der Code des Treibers verteilt sich auf die vier Module API, PCI, Controller und Codec, welche sich alle innerhalb der Kernel-Crate des D3OS (vgl. 2.2.1) befinden. Diese Module wurden in einer offenen Schichtenarchitektur strukturiert:

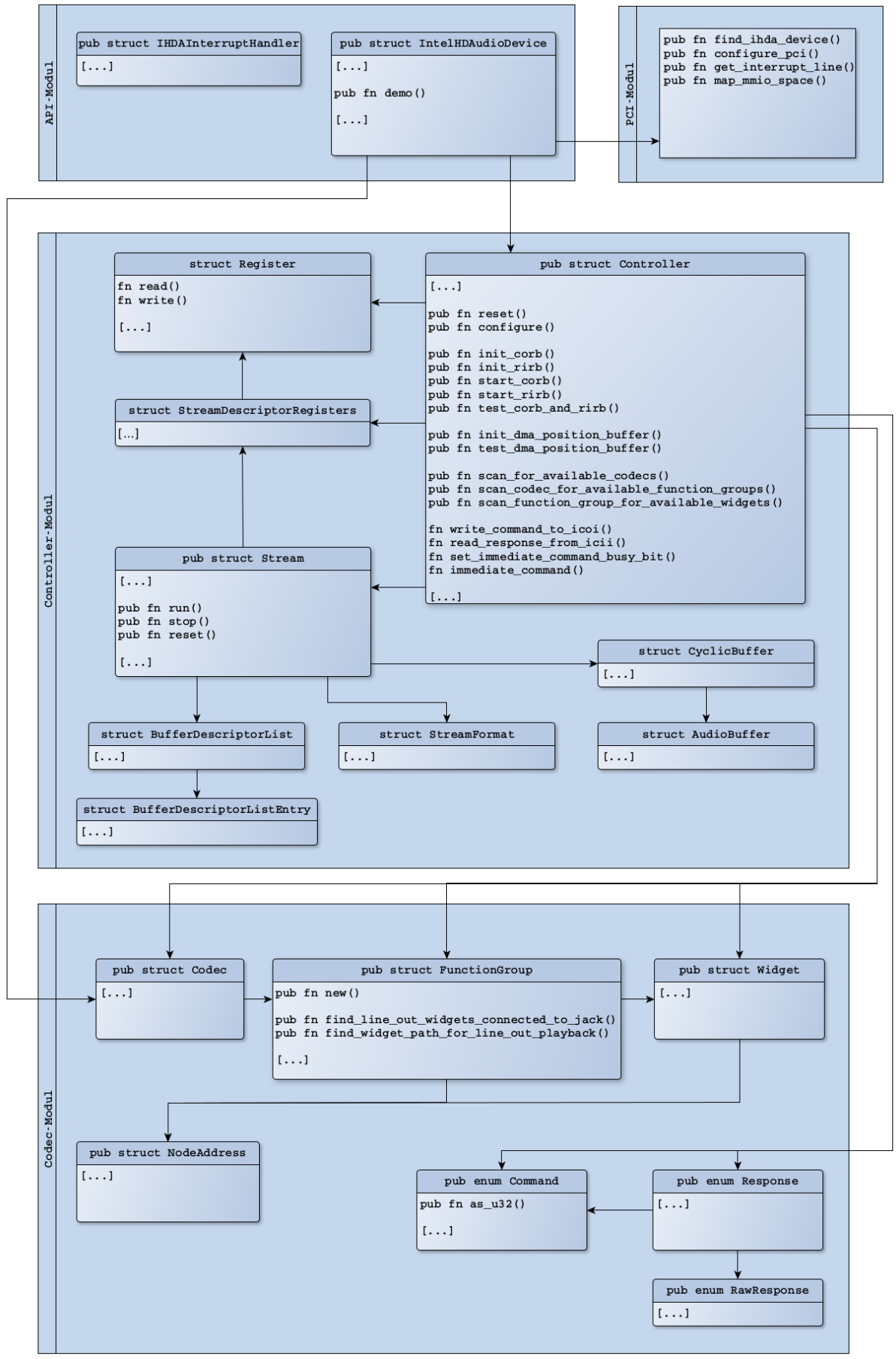


Abbildung 3.1: Schichtenarchitekur des Treibers

API-Modul: [1] In diesem Modul wird durch das Struct `IntelHDAudioDevice` und die auf diesem Struct implementierten Methoden die API definiert, über die das D3OS die Soundkarte bedient. Innerhalb der Konstruktor-Funktion dieses Structs wird die Soundkarte initialisiert und konfiguriert.
Für die Konfigurierung des PCI-Interfaces (2.6.2.1) werden dafür zunächst im PCI-Modul definierte Funktionen aufgerufen. Um den Controller zu starten und zu konfigurieren (2.6.2.2), muss dann eine Instanz des im Controller-Modul definierten Structs `Controller` erzeugt werden. Während der Ermittlung der Topologie des *Codecs* (2.6.2.3) über die Methoden dieses Structs wird zuletzt eine Instanz des im Codec-Modul definierten Structs `Codec` erstellt. Das API-Modul besitzt also Abhängigkeiten in alle drei anderen Module und steht somit an der Spitze der Schichtenarchitektur.

PCI-Modul: [2] Dieses Modul bündelt die gesamte für die Konfiguration des PCI-Interfaces (2.6.2.1) benötigte Funktionalität und besitzt keine Abhängigkeiten in eines der drei anderen Module.

Controller-Modul: [3] In diesem Modul wird das Struct `Controller` definiert, über dessen Methoden auf die *Registers* des *Controllers* zugegriffen, alle DMA-Engines, zum Beispiel für die Erstellung eines *Streams* (2.6.2.4), konfiguriert und gesteuert sowie mittels der beiden vorgestellten Mechanismen (2.6.1.2) mit dem *Codec* kommuniziert werden kann. Da ein *Stream* in der IHDA-Architektur über entsprechende *Stream Descriptor Registers* des *Controllers* erstellt und konfiguriert wird, enthält dieses Modul außerdem Implementationen der in 2.6.1.3 vorgestellten, für die Erstellung und Bedienung eines *Streams* benötigten Entitäten, insbesondere das Struct `Stream`.
Zwar wird über das Struct `Controller` mit dem *Codec* kommuniziert, jedoch befinden sich die für die Kommunikation mit dem *Codec* benötigten Implementationen aller *Commands* und *Responses* im Codec-Modul, in welches das Controller-Modul folglich Abhängigkeiten besitzt.

Codec-Modul: [4] Dieses Modul enthält Implementationen aller *Commands* und *Responses*, welche an einen *Codec* gesendet bzw. von diesem empfangen werden können. Außerdem wird dort das Struct `Codec` definiert, durch welches die Topologie des *Codecs* abgebildet werden kann. Das Codec-Modul besitzt keine Abhängigkeiten in eines der drei anderen Module.

Für die Integration des Treibers in das D3OS 3.2 musste außerdem in geringem Umfang bestehender Betriebssystem-Code angepasst bzw. erweitert werden. Der dabei hinzugefügte Code dient jedoch lediglich dazu, den Treiber zu starten und dem Betriebssystem die vom Treiber durch das Struct `IntelHDAudioDevice` bereitgestellte API zur Bedienung der Soundkarte verfügbar zu machen und ist somit nicht Teil des eigentlichen Treibers.

Es fällt auf, dass kein Modul mit Namen Link existiert und der *Link* (2.6.1.1) somit

[1] kernel/src/device/ihda_api.rs
[2] kernel/src/device/ihda_pci.rs
[3] kernel/src/device/ihda_controller.rs
[4] kernel/src/device/ihda_codec.rs

der einzige der drei Hardware-Bausteine der IHDA-Architektur ist, zu dem es keine Entsprechung im Code gibt. Dies liegt darin begründet, dass der *Link* lediglich eine serielle Verbindung zwischen *Controller* und *Codec* bereitstellt und selbst nicht konfiguriert werden kann. Welche Daten über den *Link* gesendet werden, wird also ausschließlich über *Controller* und *Codec* gesteuert, sodass keine Repräsentation des *Links* in der Software benötigt wird.

3.2 Integration in das D3OS

Das D3OS besitzt bereits Treiber für andere Hardware-Komponenten als eine Soundkarte, beispielsweise für einen Advanced Programmable Interrupt Controller (APIC) sowie eine über die PS/2-Schnittstelle angeschlossene Tastatur. Die Integration des IHDA-Treibers in das D3OS verläuft ganz analog zur Integration dieser bereits vorhandenen Treiber und beginnt innerhalb der Funktion `start`[5], welche direkt zu Beginn des Startvorgangs des Betriebssystems aufgerufen wird und die Initialisierungsprozesse verschiedener physischer und virtueller Ressourcen anstößt. In `start` wurde der Aufruf der Funktion `init_ihda`[6] hinzugefügt, wobei darauf geachtet werden musste, dass dies nach den ebenfalls in `start` stattfindenden Initialisierungen von Global Descriptor Table, Heap, APIC und PCI-Bus geschieht, da all diese Ressourcen vom IHDA-Treiber für die Initialisierung des IHDA-Geräts benötigt werden.
Die Funktion `init_ihda` erstellt eine Instanz des im API-Modul 3.3.1 definierten Structs `IntelHDAudioDevice`, indem es dessen Konstruktor-Funktion aufruft, und speichert diese Instanz in einer statischen Variable. Dadurch kann das D3OS das IHDA-Gerät über diese Instanz bis zum Herunterfahren des Betriebssystems bedienen, indem es die dafür vorgesehenen, auf dem Struct `IntelHDAudioDevice` definierten öffentlichen Methoden aufruft. Die Funktion `intel_hd_audio_device`[7] gibt eine Referenz auf diese Instanz zurück und kann aus einem beliebigen im Kernel-Adressraum laufenden Prozess aufgerufen werden. Somit ist die Soundkarte also auch aus Modulen des Kernels bedienbar, welche keine direkten Abhängigkeiten in eines der Module des Treibers besitzen.

Damit das D3OS die vier Module des Treiber erkennt, müssen diese deklariert werden[8]:

kernel/src/device/mod.rs — Rust

```rust
// [...]
pub mod ihda_api;
mod ihda_controller;
mod ihda_codec;
mod ihda_pci;
```

Im Rahmen dieser Deklaration wird auch die Sichtbarkeit der einzelnen Module festgelegt. Nur das API-Modul ist öffentlich[9] und innerhalb dieses Moduls ist `IntelHDAudioDevice` das einzige öffentliche Struct. Folglich ist dieses Struct das einzige Objekt aus allen vier Modulen des Treibers, welches außerhalb dieser Module instanziiert werden kann. Das

[5] im kernel/src/boot.rs
[6] im kernel/src/lib.rs
[7] ebenfalls im kernel/src/lib.rs
[8] Dies geschieht im kernel/src/device/mod.rs.
[9] bzw. mit dem Keyword `pub` markiert

D3OS muss also für jegliche Interaktion mit dem IHDA-Gerät die durch das Struct `IntelHDAudioDevice` gegebene API benutzen.

3.3 Detailbetrachtung der Module des Treibers

Nachdem die vier Module des Treibers und deren Abhängigkeiten untereinander bereits in 3.1 vorgestellt wurden, wird jedes dieser Module im Folgenden genauer betrachtet.

3.3.1 API-Modul

Die Struktur dieses Moduls orientiert sich an der Struktur bereits im D3OS vorhandener Treiber wie dem für einen APIC[10] oder eine PS/2-Tastatur[11]. In diesen Modulen werden jeweils zwei Structs definiert: ein API-Objekt, welches dem D3OS als Schnittstelle zur Bedienung des jeweiligen Geräts dient, und ein Interrupt Handler, welcher die Reaktionen der Software auf einen von diesem Gerät während dessen Betriebs erzeugten Hardware-Interrupt definiert. Das API-Objekt für die Bedienung des IHDA-Geräts wird durch das Struct `IntelHDAudioDevice` und der dazugehörige Interrupt Handler durch das Struct `IHDAInterruptHandler` gegeben.

Das Struct `IntelHDAudioDevice` kapselt genau eine Instanz des im Controller-Modul 3.3.3 definierten Structs `Controller` und mindestens[12] eine Instanz des im Codec-Modul 3.3.4 definierten Structs `Codec`:

ihda_driver.rs — Rust

```rust
pub struct IntelHDAudioDevice {
    controller: Controller,
    codecs: Vec<Codec>,
}

unsafe impl Sync for IntelHDAudioDevice {}
unsafe impl Send for IntelHDAudioDevice {}

// [...]
```

Die beiden Felder des öffentlichen Structs `IntelHDAudioDevice` sind privat, sodass über diese Felder nicht direkt auf *Controller* oder *Codecs* zugegriffen werden kann und stattdessen immer die dafür vorgesehenen öffentlichen API-Methoden benutzt werden müssen. Da das Struct `Controller` Zeiger zu den vom physischen Adressraum in den Kernel-Adressraum abgebildeten Speicheradressen der *Register* des *Controllers* kapselt[13], enthält auch das an das D3OS herausgegebene Struct `IntelHDAudioDevice` indirekt Zeiger zu Speicheradressen, welche außerhalb des Kernel-Adressraums ungültig wären (vgl. 2.2.2). Der Rust Compiler kann nicht wissen, dass ein `IntelHDAudioDevice` vom D3OS immer nur im Kernel-Adressraum instanziiert werden wird und diese Zeiger somit Thread-

[10] im kernel/src/device/apic.rs
[11] im kernel/src/device/ps2.rs
[12] Erinnerung: Der Einfachheit halber wurde in dieser Arbeit bisher meist von dem *Codec* gesprochen, aber grundsätzlich können auch mehrere *Codecs* an den *Link* angeschlossen sein.
[13] Dies wird in 3.3.3 näher beschrieben.

übergreifend gültig bleiben. Dass dies der Fall, ist wird dem Rust Compiler durch die Implementierung der beiden Traits `Sync` und `Send` versprochen, ohne welche das Programm nicht kompilieren würde. Die Einhaltung dieses Versprechens kann jedoch erst zur Laufzeit und nicht zur Kompilierzeit überprüft werden. Folglich ist die Implementierung dieser Traits nur in unsicherem Rust erlaubt, sodass die ensprechenden Zeilen im Code mit dem Keyword `unsafe` markiert werden müssen.
Die Konstruktor-Funktion `IntelHDAudioDevice::new` koordiniert die Initialisierung des IHDA-Gerätes und folgt dabei dem durch die Spezifikation vorgegebenen Programmiermodell 2.6.2. Im Folgenden werden die einzelnen Teile der Funktion im Detail betrachtet und mit den entsprechenden Schritten aus dem Programmiermodell in Bezug gesetzt:

ihda_driver.rs Rust

```rust
// [...]

impl IntelHDAudioDevice {
    pub fn new() -> Self {
        // 1
        let pci_bus = pci_bus();

        // 2
        let ihda_device = find_ihda_device(pci_bus);
        configure_pci(pci_bus, ihda_device);

        // 3
        let interrupt_line = get_interrupt_line(pci_bus, ihda_device);
        Self::connect_device_to_apic(interrupt_line);

        // 4
        let mmio_base_address = map_mmio_space(pci_bus, ihda_device);

        // 5
        let controller = Controller::new(mmio_base_address);

        // 6
        controller.reset();
        controller.configure();

        controller.init_corb();
        controller.init_rirb();
        controller.start_corb();
        controller.start_rirb();
        controller.test_corb_and_rirb();

        controller.init_dma_position_buffer();
        controller.test_dma_position_buffer();

        // 7
        let codecs = controller.scan_for_available_codecs();

        // 8
        Self {
            controller,
            codecs,
        }
    }

    // [...]
}
```

1. Die Funktion `pci_bus` gibt eine Referenz auf die Instanz eines Structs vom Typ `PciBus` zurück, über die auf den PCI-Bus (vgl. 2.4) zugegriffen werden kann. Sowohl `pci_bus` als auch `PciBus` werden vom D3OS bereitgestellt und geben ein anschauliches Beispiel dafür, wie auch auf das IHDA-Gerät aus beliebigen Prozessen im Kernel-Adressraum zugegriffen werden kann. Analog zum Aufruf von `pci_bus` wird zunächst die bereits in 3.2 vorgestellte Funktion `intel_hd_audio_device` aufgerufen. `intel_hd_audio_device` gibt dann eine Referenz auf eine Instanz des Structs `IntelHDAudioDevice` zurück, welches die API zur Bedienung des IHDA-Devices darstellt, genauso wie das Struct `PciBus` die API zur Bedienung des PCI-Busses darstellt.

2. Als nächstes wird ein IHDA-Gerät auf dem PCI-Bus gesucht (`find_ihda_device`, vgl. 2.6.2.1) und der PCI Configuration Space dieses Geräts konfiguriert (`configure_pci`, vgl. 2.6.2.1).

3. Die Nummer der von dem Gerät genutzten Interrupt Line wird aus dem PCI Configuration Space des Geräts ermittelt (`get_interrupt_line`, vgl. 2.6.2.1) und mit dem Interrupt Controller verbunden (`Self::connect_device_to_apic`).

4. Dann wird die MMIO-Basisadresse ermittelt und das Abbilden der *Registers* des *Controllers* vom physischen Adressraum in den virtuellen Kernel-Adressraum durchgeführt (`map_mmio_space`, vgl. 2.6.2.1), wofür ein letztes Mal auf den PCI Configuration Space zugegriffen werden muss.

5. Es wird die Konstruktor-Funktion des im Controller-Modul (3.3.3) definierten Structs `Controller` aufgerufen, wobei ihr die Adresse der *Registers* im virtuellen Kernel-Adressraum als Parameter übergeben wird. Alle folgenden Funktionsaufrufe innerhalb von `IntelHDAudioDevice::new` sind Aufrufe von Methoden auf der erzeugten Instanz. Dies spiegelt die Tatsache wieder, dass der *Controller* in der IHDA-Architektur die zentrale Programmierschnittstelle darstellt, über welche der Treiber sowohl *Controller* selbst als auch die an den *Link* angeschlossenen *Codecs* konfiguriert.

6. Der *Controller* wird gestartet und konfiguriert. Anschließend werden *CORB* und *RIRB* sowie der *DMA Position Buffer* jeweils erst initialisiert und dann gestartet (vgl. 2.6.2.2). Außerdem wird die Funktionalität dieser Buffer durch die auf dem Controller implementierten Methoden `test_corb_and_rirb` und `test_dma_position_buffer` überprüft. Diese beiden Methoden werden im Rahmen der Evaluation des Treibers in 4.2 genauer betrachtet.

7. Mithilfe des *Controllers* wird die Topologie der an den *Link* angeschlossenen *Codecs* ermittelt (`controller.scan_for_available_codecs`, vgl. 2.6.2.3). Die Anzahl der in der Variable `codecs` gespeicherten Instanzen des im Codec-Modul (3.3.4) definierten Structs `Codec` entspricht dabei genau der Anzahl an physisch vorhandenen *Codecs* auf der Soundkarte.

8. Zuletzt wird eine Instanz des Structs `IntelHDAudioDevice` erzeugt und an den Aufrufer der Konstruktor-Funktion zurückgegeben, sodass der Aufrufer diese Instanz als API zur Bedienung der Soundkarte benutzen kann.

3.3.2 PCI-Modul

In diesem Modul werden vier Funktionen definiert, welche alle gemeinsam haben, dass ihnen bei ihrem Aufruf eine Referenz auf ein Struct vom Typ `PciBus` übergeben wird, über das sowohl auf den gesamten PCI-Bus als auch auf den PCI Configuration Space einzelner PCI-Geräte zugegriffen werden kann.
Die Funktion `find_ihda_device` sucht den PCI-Bus nach einem vom Treiber unterstützten IHDA-Gerät ab. Die anderen drei Funktionen lassen sich jeweils einem Schritt der Initialisierung des PCI-Interfaces (2.6.2.1) zuordnen. Die Funktion `configure_pci` setzt die beiden Bits im PCI Configuration Space des IHDA-Geräts (2.6.2.1), die Funktion `get_interrupt_line` ermittelt die von dem IHDA-Gerät genutzte Interrupt Line aus dem PCI Configuration Space (2.6.2.1) und die Funktion `map_mmio_space` bildet die MMIO-Basisadresse der *Registers* des *Controllers* vom physischen Adressraum in den virtuellen Kernel-Adressraum ab und gibt die virtuelle Startadresse des allozierten Speicherblocks[14] zurück (2.6.2.1).

3.3.3 Controller-Modul

Während das im API-Modul (3.3.1) definierte Struct `IntelHDAudioDevice` die Schnittstelle zwischen D3OS und Treiber darstellt, bildet das in diesem Modul definierte Struct `Controller` die Schnittstelle zwischen Treiber und *Controller*. Um eine Instanz des Structs `Controller` zu erzeugen, muss der Konstruktor-Funktion lediglich die MMIO-Basisadresse im Kernel-Adressraum, also die Adresse, an welche die *Registers* abgebildet wurden, als Parameter übergeben werden.[15] Aus dieser Adresse können die Adressen aller anderen *Registers* abgeleitet werden, da diese jeweils ein festes, durch die Spezifikation vorgegebenes Offset zur MMIO-Basisadresse besitzen.
Jedes *Register* des *Controllers* wird durch eine Instanz des Structs `Register` repräsentiert, welches einen Zeiger auf die individuelle Adresse des *Registers* kapselt und Methoden für lesende und schreibende Zugriffe auf das *Register* bereitstellt. Diese Zugriffe sind inhärent unsicher und bedürfen somit der Benutzung von unsicherem Rust (vgl. 2.1.1). Die auf dem Struct `Register` definierten Methoden `read` und `write` enthalten jeweils lediglich eine in unsicherem Rust geschriebene Zeile mit dem Speicherzugriff und sind wiederum selbst sichere Funktionen.[16]
Jede einzelne DMA-Engine zur Übertragung eines *Streams* besitzt ihre eigenen neun *Stream Descriptor Registers*. Jedes dieser *Stream Descriptor Registers* wird, wie auch alle anderen *Registers* des *Controllers*, durch eine Instanz des Structs `Register` repräsentiert und die neun *Stream Descriptor Registers* einer einzelnen DMA-Engine werden gemeinsam in dem Struct `StreamDescriptorRegisters` gekapselt. Da die Anzahl der in einem *Controller* verbauten DMA-Engines von Gerät zu Gerät unterschiedlich ist, müssen während der Instanziierung des Structs `Controller` innerhalb seiner Konstruktor-Funktion zunächst die Anzahl an Input-, Output- und bidirektionalen DMA-Engines aus dem *Global Capacities Register* (vgl. 2.5) ermittelt und anschließend genauso viele Instanzen des Structs `StreamDescriptorRegisters` erstellt werden, wie Engines vorhanden sind. Alle Felder des Structs `Controller` sind entweder Structs vom Typ `Register` oder vom Typ

[14] Momentan bildet das D3OS Adressen aus dem physischen Adressraum immer eins zu eins in den virtuellen Kernel-Adressraum ab.
[15] Erinnerung: An dieser Adresse befindet sich das *Global Capabilities Register* (vgl. 2.5).
[16] Sie bilden somit eine sichere API zur Ausführung des unsicheren Codes (vgl. 2.1.1).

`StreamDescriptorRegisters`.
Die meisten der auf dem Struct `Controller` definierten Methoden führen einen einzelnen atomaren lesenden oder schreibenden Zugriff auf genau ein *Register* aus, wie beispielsweise die Methode `set_immediate_command_busy_bit`, welche das *ICB Bit* an Position 0 im *ICIS Register* setzt (vgl. 2.6.1.2). Diese atomaren Methoden nutzen für diese unsicheren Zugriffe wiederum die im vorangegangenen Abschnitt vorgestellten sicheren Abstraktionen, welche durch die auf dem Struct `Register` definierten Methoden bereitgestellt werden, und sind somit selbst sicher.
Andere auf dem Struct `Controller` definierte Methoden erledigen komplexere Aufgaben, während derer meist mehrere andere atomare Methoden des Structs aufgerufen werden. Ein Beispiel dafür ist die Methode `immediate_command`, welche eine Instanz des Structs `Command` als Parameter übergeben bekommt, dieses über die *ICIO Registers* versendet und die von der adressierten *Node* erhaltene *Response* in Form einer Instanz des Structs `Response` zurückgibt.[17] Dafür ruft sie die atomaren Methoden `write_command_to_icoi`, `set_immediate_command_busy_bit` und `read_response_from_icii` auf:

[17] Die Structs `Command` und `Response` werden in 3.3.4 vorgestellt.

ihda_controller.rs Rust

```rust
impl Controller {
    // [...]

    fn write_command_to_icoi(&self, command: Command) {
        self.icoi.write(command.as_u32());
    }

    // [...]

    fn read_response_from_icii(&self) -> u32 {
        self.icii.read()
    }

    // [...]

    fn set_immediate_command_busy_bit(&self) {
        self.icsts.set_bit(0);
    }

    // [...]

    fn immediate_command(&self, command: Command) -> Response {
        self.write_command_to_icoi(command);
        self.set_immediate_command_busy_bit();

        // [...]

        let raw = RawResponse::new(self.read_response_from_icii());
        Response::new(raw, command)
    }
}
```

Alle Felder des Structs `Controller` sowie die meisten auf dem Struct definierten Methoden und insbesondere alle atomaren Methoden sind privat, damit Code, welcher einen `Controller` instanziiert, nicht über diese Methoden willkürlich auf die *Registers* des *Controllers* zugreifen kann. Stattdessen müssen die bereitgestellten öffentlichen Methoden benutzt werden, welche sicherstellen, dass der *Controller* im Sinne der Spezifikation bedient wird.

Für die Ermittlung der Topologie der an den *Link* angeschlossenen *Codecs* (vgl. 2.6.2.3) müssen die einzelnen *Nodes* dieses *Codecs* mittels geeigneter *Commands* nach ihren Eigenschaften und Kapazitäten befragt werden, wofür ein ein *Controller* benötigt wird (vgl. 2.6.1.2). Zu diesem Zweck sind auf dem Struct `Controller` die Methoden `scan_for_available_codecs`, `scan_codec_for_available_function_groups` und `scan_function_group_for_available_widgets` definiert, innerhalb welcher die für die Instanziierung der im Codec-Modul definierten Structs `Codec`, `FunctionGroup` und `Widget`

benötigten *Responses* von den *Codecs* angefordert werden[18].

Da der *Controller* eines IHDA-Geräts alle für die Übertragung von *Streams* benötigten DMA-Engines verwaltet (vgl. 2.6.1.3), wird im Controller-Modul neben dem zentralen Struct `Controller` ebenfalls das Struct `Stream` definiert. Innerhalb der Konstruktor-Funktion `Stream::new` wird ein Stream wie in 2.6.2.4 beschrieben erstellt, indem Audiobuffer alloziert und die *Stream Descriptor Registers* der benutzten DMA-Engine konfiguriert werden. Während dieses Prozesses werden außerdem Instanzen der ebenfalls in diesem Modul definierten Structs `BufferDescriptorList`, `BufferDescriptorListEntry`, `AudioBuffer`, `CyclicBuffer` und `StreamFormat` erzeugt, welche jeweils eine der in 2.6.1.3 vorgestellten Entitäten repräsentieren und benötigt werden, um eine Instanz des Structs `Stream` zu erzeugen. Wie auch bei den Zugriffen auf die *Registers* des *Controllers* sind Zugriffe auf andere DMA-Bereiche wie die *BDL* oder die Audiobuffer inhärent unsicher und bedürfen somit der Benutzung von unsicherem Rust (vgl. 2.1.1). Auf solch einer Instanz können Methoden aufgerufen werden, um den durch diese Instanz repräsentierten *Stream* zu steuern. Beispielsweise startet die Methode `run` den *Stream*, die Methode `stop` stoppt ihn wieder und durch die Methode `reset` werden alle *Stream Descriptor Registers* des *Streams* zurückgesetzt.

3.3.4 Codec-Modul

Dieses Modul enthält die Implementationen der für die Kommunikation mit den *Codecs* benötigten *Commands* und *Responses* (vgl. 2.6.1.2) und definiert zudem die Structs `Codec`, `FunctionGroup`, `Widget` und `NodeAddress`, welche für die Abbildung der Topologie eines *Codecs* (vgl. 2.6.2.3) benötigt werden.

Jede Variante des Enums `Command` entspricht genau einem in der Spezifikation definierten *Command* und kapselt die aus Adresse des *Codecs* und *Node ID* bestehende Adresse der *Node*, an welche das *Command* gesendet werden soll, und den *Payload* des *Commands*, während der vierte Bestandteil (vgl. 2.6.1.2) des *Commands*, die *Verb ID*, implizit durch die Variante des Enums gespeichert wird:

[18] Dieser Instanziierungsprozess wird in 3.3.4 näher erläutert.

ihda_codec.rs Rust

```rust
pub enum Command {
    GetParameter(NodeAddress, Parameter),
    GetConnectionSelect(NodeAddress),
    SetConnectionSelect(NodeAddress, SetConnectionSelectPayload),
    GetConnectionListEntry(NodeAddress, GetConnectionListEntryPayload),
    GetAmplifierGainMute(NodeAddress, GetAmplifierGainMutePayload),
    SetAmplifierGainMute(NodeAddress, SetAmplifierGainMutePayload),
    GetStreamFormat(NodeAddress),
    SetStreamFormat(NodeAddress, SetStreamFormatPayload),
    GetChannelStreamId(NodeAddress),
    SetChannelStreamId(NodeAddress, SetChannelStreamIdPayload),
    GetPinWidgetControl(NodeAddress),
    SetPinWidgetControl(NodeAddress, SetPinWidgetControlPayload),
    GetEAPDBTLEnable(NodeAddress),
    SetEAPDBTLEnable(NodeAddress, SetEAPDBTLEnablePayload),
    GetConfigurationDefault(NodeAddress),
    GetConverterChannelCount(NodeAddress),
    SetConverterChannelCount(NodeAddress, SetConverterChannelCountPayload),
}
```

Die auf diesem Enum definierte Methode `as_u32` übersetzt die gekapselten Informationen in das vom *Controller* akzeptierte Format eines 4-Bytes-Pakets (vgl. 2.6.1.2).
Jede Variante des Enums `Response` entspricht genau einer in der Spezifikation definierten *Response*. Die als Reaktion auf das Versenden eines *Commands* erhaltene *Response* eines *Codecs* ist zunächst ebenfalls ein opakes 4-Bytes-Paket und kann nur dann sinnvoll interpretiert werden, wenn das dazu gehörende *Command* bekannt ist. `Response::new` bekommt deshalb sowohl eine `RawResponse`, welches das erhaltenen 4-Bytes-Paket kapselt, und ein `Command` übergeben, dessen Variante bestimmt, wie diese 32-Bit-Folge interpretiert werden soll.
Während alle *Commands* und *Responses* im Codec-Modul definiert sind, befinden sich die Implementierungen der beiden Schnittstellen zur Kommunikation mit den *Codecs* (*CORB* und *RIRB* sowie die *ICIO Register*) im Controller-Modul (3.3.3), da diese Schnittstellen nur über den *Controller* des IHDA-Geräts konfiguriert und betrieben werden können. Der im Controller-Modul implementierte *Controller* bedient sich also der hier im Codec-Modul definierten *Commands* und *Responses* und trägt die Verantwortung für die Zuordnung einer erhaltenen *Response* zu einem versendeten *Command*.

Die drei Structs `Codec`, `FunctionGroup` und `Widget` stehen in einem aus der Architektur eines *Codecs* (vgl. 2.6.1.1) abgeleitenen Verhältnis zueinander: Eine Instanz des Structs `Codec` enthält mindestens eine Instanz vom Struct `FunctionGroup`, welche wiederum mehrere Instanzen des Structs `Widget` enthält. Dies ist die Definition des Structs `Codec` sowie dessen Konstruktor-Funktion:

ihda_codec.rs Rust

```rust
pub struct Codec {
    codec_address: CodecAddress,
    vendor_id: VendorIdResponse,
    revision_id: RevisionIdResponse,
    function_groups: Vec<FunctionGroup>
}

impl Codec {
    pub fn new(
        codec_address: CodecAddress,
        vendor_id: VendorIdResponse,
        revision_id: RevisionIdResponse,
        function_groups: Vec<FunctionGroup>
    ) -> Self {
        Codec {
            codec_address,
            vendor_id,
            revision_id,
            function_groups,
        }
    }
}
```

Die innerhalb der Konstruktor-Funktion zu erwartende Konstruktion der Instanz des Structs wurde also vollständig in das Controller-Modul ausgelagert, wo sie innerhalb von auf dem Struct `Controller` implementierten Methoden stattfindet.[19] Diese Auslagerung liegt im Programmiermodell begründet, gemäß dem die Ermittlung der Topologie eines *Codecs* über den *Controller* erfolgen muss (vgl. 2.6.2.3). Instanzen des Structs `Codec`[20] können also erst erstellt werden, nachdem der *Controller* die für ihre Instanziierung benötigten *Responses* bereitgestellt hat.
Während der Befragung der *Codecs* wird einerseits die Topologie der *Codecs* ermittelt und andererseits werden weitere *Parameters* der einzelnen *Nodes* abgefragt, mit deren Hilfe zum Beispiel herausgefunden werden kann, ob eine *Node* einen oder mehrere konfigurierbare Verstärker besitzt. Die Speicherung all dieser Informationen im Rahmen der Ermittlung der Topologie des *Codecs* wird nicht direkt durch das Programmiermodell (2.6.2) vorgegeben, da diese Informationen auch später während des Betriebs der Soundkarte jederzeit eingeholt werden können. Jedoch erlaubt dieses initiale Speichern möglichst vieler statischer Informationen über die einzelnen *Nodes* eines *Codecs*, Suchoperationen innerhalb des *Codecs* auszuführen, ohne weitere *Commands* über den *Controller* an den *Codec* senden zu müssen. Folglich können beispielsweise die für die Konfigurierung des *Codecs* zur Audiowiedergabe (vgl. 2.6.2.5) genutzten Methoden `find_line_out_pin_widgets_connected_to_jack`, welche im *Codec* ein für die Audioausgabe benötigtes *LOPW* (vgl. 2.6.1.1) sucht, und

[19] Dies sind die Methoden `scan_for_available_codecs`, `scan_codec_for_available_function_groups` und `scan_function_group_for_available_widgets`.
[20] und auch der Structs `FunctionGroup` und `Widget`

`find_widget_path_for_line_out_playback`, welche einen Pfad zwischen diesem *LOPW* und einem ebenfalls benötigten *AOCW* ermittelt, auf dem Struct `FunctionGroup` implementiert werden und müssen nicht ebenfalls in das Controller-Modul auslagert werden.

Kapitel 4

Evaluation

Da jeder *Stream* in der IHDA-Architektur eine feste Samplerate und Bittiefe besitzt, werden die durch diesen *Stream* übermittelten Audiodaten mit einer konstanten Rate über den *Link* geschickt. Wichtig für eine korrekte Übertragung ist also die Aufrechterhaltung eines konstanten Stroms an Samples durch *Controller* und Treiber, und es kommt nicht darauf an, möglichst viele Samples möglichst schnell über den Link zu schicken. Folglich ergibt das Erstellen von Benchmarks im Rahmen dieser Treiberevaluation keinen Sinn. Stattdessen wird in 4.2 verifiziert, dass der Treiber der in 1.1 formulierten Zielsetzung gerecht wird, indem er dem D3OS durch das Struct `IntelHDAudioDevice` (vgl. 3.3.1) eine API zur Verfügung stellt, über welche es ein digitales Audiosignal in Form eines *Streams* korrekt über einen analogen Ausgang der in den Testrechner verbauten integrierten Soundkarte wiedergeben kann. Doch zunächst werden in 4.1 die größten Herausforderungen im Verlauf der Treiberentwicklung beschrieben und zur Bewältigung dieser Herausforderungen getroffenene Entscheidungen erläutert.

4.1 Herausforderungen während der Treiberentwicklung

Komplexität und Offenheit der Intel HD Audio Architektur: Die größte Herausforderung während der gesamten Treiberentwicklung war die Bewältigung der durch die 223-seitige IHDA-Spezifikation[2] beschriebenen Komplexität eines IHDA-Geräts. Während in 2.6 versucht wurde, die Funktionsweise einer IHDA-Soundkarte in einem Top-Down-Prozess zu erklären, beschreibt die Spezifikation einzelne Aspekte von Architektur und Programmiermodell oft nur kurz und springt dann zu einem völlig anderen Aspekt. Zudem richtet sich die Spezifikation nicht nur an Treiberentwickler, sondern auch an Hardware-Hersteller, wodurch zwischen den für die Treiberentwicklung relevanten Informationen auch immer wieder irrelevante Informationen auftauchen. Ob eine Information relevant ist, ist dabei häufig nicht sofort klar.
Außerdem scheint die Treiberprogrammierung für IHDA-Geräte generell kein populäres Thema zu sein. Somit konnten während der Recherche zu dieser Arbeit keine wissenschaftlichen Texte und nur wenige andere Quellen zu IHDA gefunden werden. Die wichtigsten beiden Hilfen beim Verständnis der Spezifikation waren ein Wiki-Eintrag und eine Forendiskussion auf der Seite osdev.org, einer Online-Community für Betriebssystementwickler. Zudem wurden im Rahmen der Recherche zu dieser Arbeit kurz sowohl der in C geschriebene IHDA-Treiber für Linux[63] als auch der in Rust geschriebene Treiber für das Redox OS[64] betrachtet. Da die beiden Treiber jedoch bereits sehr viel komplexer sind als der

selbst entwickelte, wurde die Entscheidung getroffen, die zur Verfügung stehende Zeit statt in das Studium anderer Treiber und Betriebssysteme besser in das Studium der Spezifikation zu investieren, sodass die gefundenen Vorlagen während der Treiberentwicklung nicht weiter berücksichtigt wurden.

Während der Treiberentwicklung wurde versucht, den Treiber so universell wie möglich zu halten, und in weiten Teilen ist dies auch gelungen. Somit sollten die Initialisierung des PCI-Interfaces (2.6.2.1), Start und Konfigurierung des *Controllers* (2.6.2.2), die Ermittlung der Topologie aller an den Link angeschlossenen *Codecs* (2.6.2.3) sowie die Erstellung eines *Streams* (2.6.2.4) auf jedem beliebigen IHDA-Gerät funktionieren. Lediglich die Konfigurierung des *Codecs* für die Audiowiedergabe (2.6.2.5) funktioniert ausschließlich für den in die integrierte Soundkarte des für die Entwicklung genutzten Testrechners verbauten *Codec*[1]. Grund dafür ist die aus der modularen Architektur eines *Codecs* (vgl. 2.6.1.1) resultierende Freiheit eines Herstellers beim Entwurf eines *Codecs*.
Einen Algorithmus zu entwerfen, welcher universell die Topologie aller an einen *Link* angeschlossenen *Codecs* ermittelt, ist nicht sehr schwierig und auch im Rahmen der Treiberentwicklung gelungen. Jedoch ist es wesentlich komplizierter einen Algorithmus zu finden, welcher universell auf jedem beliebigen *Codec* einen Ausgabepfad sowohl findet als auch konfiguriert (vgl. 2.6.2.5). Diese Aufgabe ist aus mehreren Gründen nicht trivial. Zunächst können in einem *Codec* theoretisch beliebig viele individuell zu konfigurierende *Widgets* zwischen *AOCW* und *LOPW* liegen, wobei auch *Vendor Defined Widgets* erlaubt sind, welche nicht durch die Spezifikation beschrieben werden. Außerdem kann es Zyklen im Verknüpfungsgraphen der *Widgets* geben (vgl. Abb. 6.1), welche ein Algorithmus erkennen muss, um nicht in eine Endlosschleife zu laufen. Schließlich muss der Algorithmus zudem entscheiden können, welches *LOPW* für die Audiowiedergabe benutzt werden soll, falls mehrere *LOPWs* in einen *Codec* verbaut sind.[2]
Aufgrund dieser Hürden überprüft der Treiber in seiner Version zum Zeitpunkt der Fertigstellung dieser Arbeit vor der Konfigurierung eines *Codecs* für die Audiowiedergabe zunächst, ob *Vendor ID* und *Device ID* des *Codecs* identisch zu denen des in der integrierten Soundkarte des Testrechners verbauten *Codecs*[3] sind und bricht ansonsten die Konfigurierung ab [4], was in dieser Version des Treibers noch gleichbedeutend mit einem kompletten Abbruch des Startvorgangs des Betriebssystems ist und in zukünftigen Versionen durch eine weniger radikale Art der Fehlerbehandlung ersetzt werden sollte. In seiner Version zu Fertigstellung dieser Arbeit unterstützt der Treiber also genau ein IHDA-Soundkarten-Modell.

Testen auf virtualisierter Hardware: Das Testen auf virtualisierter Hardware ist wie in 2.3 beschrieben sehr komfortabel und schnell im Vergleich zum Testen auf physischer

[1] Übrigens besitzen sowohl ein IHDA-Gerät als auch alle darauf verbauten Codecs ihre eigenen *Vendor IDs* und *Device IDs*. Theoretisch können in einem IHDA-Gerät also Codecs verschiedener Hersteller verbaut sein.

[2] Dies ist bei Soundkarten mit mehr als einem analogen Ausgang, zum Beispiel für die Wiedergabe von Surround-Sound, der Fall.

[3] Dies sind *Vendor ID* 0x10EC (Realtek Semiconductor Corporation) und *Device ID* 0x280.

[4] Auch der bereits angesprochene IHDA-Treiber für Linux enthält eine lange Auflistung an *Vendor IDs* und *Device IDs*, sodass der selbst entwickelte Treiber nicht der einzige zu sein scheint, welcher erweitert werden muss sobald er ein IHDA-Gerät unterstützen soll, welches er noch nicht kennt.

Hardware. Jedoch stellte sich während der Entwicklung des Treibers heraus, dass die Virtualisierung der Soundkarte und insbesondere die Emulierung einiger DMA-Engines der Soundkarte in der genutzten QEMU Version 8.2.1 nicht korrekt funktionierte. Einerseits reagierte der vom *Controller* verwaltete *CORBWP* nicht auf durch den Treiber gemachte Aktualisierungen des *CORBWP* (vgl. 2.6.1.2) und andererseits hängte sich die virtuelle Maschine komplett auf, sobald eine DMA-Engine für die Übertragung eines *Streams* gestartet wurde (vgl. 2.6.1.3). Da der geschriebene Code jedoch korrekt auf dem physischen Testrechner funktionierte, wurde etwa zur Mitte der dreimonatigen Entwicklungszeit die pragmatische, aber durchaus schmerzliche Entscheidung getroffen, das Testen in der zweiten Hälfte der Entwicklungsperiode hauptsächlich auf dem physischen Testrechner durchzuführen und der Ursache des Problems in QEMU nicht weiter nachzuspüren.
Dabei soll nicht unterschlagen werden, dass einige Dinge auch in QEMU funktionierten. Treiber und *Controller* konnten grundsätzlich auch auf dem virtualisierten Testechner über die in den virtuellen Kernel-Adressraum abgebildeten *Registers* kommunizieren, und über die darin enthaltenen *ICIO Register* konnte der Treiber erfolgreich *Commands* an den emulierten *Codec* schicken und erhielt daraufhin sogar sinnvolle *Responses* zurück. Bemerkenswert ist, dass dieser korrekt funktionierende DMA-Bereich mit den *Registers* der einzige aus dem gesamten Programmiermodell ist, welcher nicht vom Treiber selbst alloziert werden muss[5].
Insgesamt kann zusammengefasst werden, dass das Testen auf einem virtualisierten Rechner insbesondere zu Beginn der Treiberentwicklung sehr hilfreich dafür war, einen Überblick über die Soundkarte zu gewinnen. Sobald allerdings vom Treiber allozierte DMA-Bereiche zum Beispiel für *CORB* und *RIRB* oder aber den Betrieb eines *Streams* benutzt werden mussten, zeigte die emulierte Soundkarte inkorrektes Verhalten, welches auf physischer Hardware nicht zu beobachten war.

Das noch junge D3OS: Im Rahmen der Treiberentwicklung für das noch junge D3OS musste an manchen Stellen zunächst dem Betriebssystem selbst Funktionalität hinzugefügt werden, bevor die eigentliche, durch die IHDA-Spezifikation vorgegebene Funktionalität implementiert werden konnte. Beispielsweise stellen die Funktionen im PCI-Modul des Treibers (vgl. 3.3.2) eigentlich allgemeine Betriebssystemfunktionalität und somit keine exklusive Funktionalität eines IHDA-Geräts bereit, denn das Setzen von Bits im PCI Configuration Space eines PCI-Geräts oder das Ermitteln der Interrupt Line muss auch für andere Geräte als IHDA-Soundkarten vollzogen werden. Statt im PCI-Modul des Treibers könnten diese Funktionen zum Beispiel besser als Methoden des in 3.3.1 vorgestellten Structs `PciBus` implementiert werden.[6] Infolge dessen würde das PCI-Modul gar nicht mehr benötigt werden.
Ein weiteres Beispiel stellt die Funktion `alloc_no_cache_dma_memory` im Controller-Modul dar, welche zunächst einen Speicherblock alloziert und anschließend für alle Seiten dieses Speicherblocks das No-Cache-Flag setzt.[7] Dieses Flag muss für alle vom *Controller* genutzten DMA-Bereiche gesetzt werden, da vom Treiber gemachte Änderungen in die-

[5] Erinnerung: Dieser DMA-Bereich wird bereits bei der Initialisierung des PCI-Busses während des Systemstarts alloziert, und diese Adresse muss vom Treiber aus dem PCI Configuration Space der Soundkarte ermittelt werden.

[6] Dann müsste den Methoden auch nicht jeweils eine Instanz des Structs `PciBus` als Parameter übergeben werden.

[7] Das D3OS konnte auch vorher schon Speicherblöcke allozieren, jedoch war für keine der allozierten Seiten das No-Cache-Flag gesetzt.

sen DMA-Bereichen immer sofort in den Arbeitsspeicher geschrieben werden müssen und nicht etwa aus Gründen der Effizienz nur in die Caches des Arbeitsspeichers geschrieben werden dürfen. Der *Controller* kann nämlich lediglich auf den Arbeitsspeicher und nicht auf dessen Caches zugreifen und würde vom Treiber gemachte Änderungen in den Caches gar nicht mitbekommen und folglich auch nicht auf diese Änderungen reagieren. Diese Art der Speicherallozierung ist, wie auch schon die durch die Funktionen des PCI-Moduls gegebene Funktionalität, nicht exklusiv für IHDA-Geräte nützlich, sondern für jegliche Geräte, die DMA benutzen. Somit ist auch diese Funktion innerhalb der Module des Treibers etwas fehl am Platze.
Sowohl die Funktionen im PCI-Modul als auch die Funktion `alloc_no_cache_dma_memory` wurden letztendlich an ihren nicht perfekt geeigneten Orten gelassen und nicht etwa auf andere Module des Betriebssystems verteilt, damit besser ersichtlich bleibt, welcher Code im Rahmen dieser Treiberentwicklung vom Autor selbst geschrieben wurde und welcher Code bereits vor Beginn der Treiberentwicklung im D3OS existierte.
Zuletzt sei noch erwähnt, dass das Setzen des No-Cache-Flags schließlich doch nicht so funktionierte wie erwartet. Obwohl dieses Flag für die Audiobuffer eines *Streams* gesetzt wurde, konnte dieser *Stream* nur dann erfolgreich übertragen werden, wenn nach dem Schreiben in diese Buffer zusätzlich alle Caches des Arbeitsspeicher manuell gespült wurden.[8] Der Grund dafür konnte bis zum Ende der Entwicklungsperiode nicht gefunden werden und versteckt sich womöglich tiefer in der Speicherverwaltung des D3OS. Dieses Beispiel zeigt, dass neben der allgegenwärtigen Komplexität der Intel HD Audio Architektur immer wieder auch die Komplexität des D3OS zutage trat und bewältigt werden musste.

4.2 Funktionalität des Treibers

Um die Funktionalität des Treibers zu überprüfen, wurden drei Testfunktionen geschrieben, welche im Rahmen der Initialisierung der Soundkarte aufgerufen werden und zusammen aufzeigen, dass das in 2.6.2 vorgestellte Programmiermodell prinzipiell korrekt implementiert wurde, dass der *Controller* erfolgreich auf alle zu den fünf verschiedenen in 2.6.3 aufgelisteten Zwecken auf die vom Treiber allozierten DMA-Bereiche zugreifen kann, und dass der Treiber dem D3OS schließlich auch die in 1.1 geforderte API zur Verfügung stellt, über welche das Betriebssystem die Wiedergabe eines Audiosignal über einen analogen Ausgang der Soundkarte starten kann.

In 3.3.1 wurde bereits erwähnt, dass das im Controller-Modul definierte Struct `Controller` die beiden Methoden `test_corb_and_rirb` und `test_dma_position_buffer` implementiert. Diese beiden Methoden werden innerhalb der Konstruktor-Funktion des Structs `IntelHDAudioDevice` aufgerufen, nachdem die dazugehörigen Buffer initialisiert und gestartet wurden.
Innerhalb von `test_corb_and_rirb` werden zwei identische und gültige *Commands*[9] in den *CORB* gelegt und der *CORBWP* entsprechend um 2 inkrementiert (vgl. 2.6.1.2). Daraufhin sendet der *Controller* die beiden *Commands* an den *Codec* und legt die erhal-

[8] Dies geschah über das Senden des Assembler-Befehls `wbinvd` an den Prozessor.
[9] Es wird jeweils der *Parameter* namens *Vendor ID* von der Root Node des Codecs angefragt.

tenen *Responses* in den *RIRB*. Da die gesendeten *Commands* identisch waren, müssen auch die beiden *Responses* identisch sein. Da die *Commands* zudem gültig waren, dürfen die *Responses* nicht gleich 0 sein.[10] Falls eine dieser Bedingungen verletzt ist, wird die Initialisierung der Soundkarte abgebrochen.
Innerhalb von `test_dma_position_buffer` wird zunächst ein Dummy-*Stream* erstellt, an die erste verfügbare Output-DMA-Engine des *Controllers* gebunden und anschließend gestartet. Dann wird kurz hintereinander die DMA Position dieser Engine aus dem *DMA Position Buffer* gelesen und überprüft, dass die beiden gelesenen Werte unterschiedlich und ungleich 0 sind, woraus folgt, dass die Offsets der Adressen innerhalb des *Cyclic Buffers* des *Streams* durch den *Controller* aktualisiert werden. Zudem wird verifiziert, dass die DMA Positionen aller anderen Output-DMA-Engines[11] weiterhin gleich 0 sind, da diese ja nicht gestartet wurden. Falls eine dieser Bedingungen verletzt ist, wird die Initialisierung der Soundkarte abgebrochen.

Die dritte Testfunktion heißt `demo` und wurde nicht auf dem Struct `Controller`, sondern als öffentliche Methode auf dem im API-Modul definierten Struct `IntelHDAudioDevice` implementiert.
Innerhalb von `demo` wird zunächst ein *Stream Format* mit einem *Channel* (mono), einer Samplerate von 48 kHz und einer Bittiefe von 16 Bit definiert und ein *Stream* mit diesem Format erstellt. Für diesen *Stream* werden zwei Audiobuffer mit jeweils 65536 Bytes Größe alloziert, welche bei dem gewählten *Stream Format* jeweils Platz für 65536 Bytes / 16 Bits = 32768 Samples bieten. Dann wird in beide Audiobuffer dasselbe Signal in Form einer Sägezahnschwingung mit einer Wellenlänge von genau 64 Samples gelegt.[12] Dies bedeutet, dass bei der korrekten Wiedergabe dieses Audiosignals bei einer Samplerate von 48 kHz ein Ton mit einer Grundfrequenz von genau 48 kHz / 64 = 750 Hz zu hören sein und dieser das charakteristische Obertonspektrum einer Sägezahnschwingung (vgl. 2.3b) besitzen sollte. Als nächstes werden die *Widget Nodes* auf dem Ausgabepfad konfiguriert. Im konkreten Fall des in die integrierte Soundkarte des Testrechners verbauten *Codecs* müssen ein *AOCW*, ein *MW* und ein *LOPW* konfiguriert werden, indem deren eingebaute Verstärker aktiviert, Stummschaltungen entfernt und die Wiedergabelautstärke eingestellt werden.[13] Zuletzt wird der *Stream* gestartet, sodass die für diesen *Stream* zuständige DMA-Engine mit der Übertragung der Samples beginnt. Da die DMA-Engine die Audiobuffer immer wieder zyklisch durchläuft und die Daten in den Audiobuffern nach dem Start des *Streams* nie ausgetauscht werden (vgl. 2.6.1.3), ist zu erwarten, dass der wiedergegebene Ton mit einer Frequenz von 750 Hz in konstanter Lautstärke hörbar bleibt, solange das D3OS nicht beendet wird.

[10] Die *Response* auf ein ungültiges *Command* ist immer gleich 0. Solch eine *Response* erhält man zum Beispiel, wenn man bei einer anderen *Node* als der *Root Node* den *Parameter* namens *Vendor ID* anfragt.

[11] Der *Controller* der in den Tesrechner verbauten Soundkarte besaß vier Output-DMA-Engines (und vier Input-DMA-Engines und keine bidirektionalen DMA-Engines).

[12] Ein Sample im PCM-Format wird als Signed Integer gespeichert, sodass ein Sample bei einer Bittiefe von 16 Bit Werte zwischen -32768 und 32767 annehmen kann. Der Sägezahn wurde konstruiert, indem dem i-ten Sample des *Streams* der Wert -32768 + ((i modulo 64) $*$ 1024) zugewiesen wurde.

[13] Erinnerung: Dies geschieht über das Verändern entsprechender *Controls* der *Nodes* mittels geeigneter *Commands* (vgl. 2.6.2.5). Diese Konfigurierung funktioniert in der vorgestellten Version des Treibers nur für *Codecs*, welche gleiche *Vendor ID* und *Device ID* wie der *Codec* der integrierten Soundkarte des Testrechners besitzen (vgl. 4.1).

In 3.2 wurde erklärt, wie das D3OS eine Instanz des Structs `IntelHDAudioDevice` erzeugt, um die Soundkarte über die auf diesem Struct definierten öffentlichen Methoden zu bedienen. Genau solch eine Methode ist `demo`. Um zu zeigen, dass diese API tatsächlich funktioniert, wurde der ebenfalls in 3.2 vorgestellten Funktion `start`, welche außerhalb der Module des Treibers definiert ist, unmittelbar hinter dem Aufruf der Funktion `init_ihda`, welche die Initialisierung der Soundkarte anstößt, der Aufruf der Methode `demo` hinzugefügt.[14]
Wurde das D3OS nun auf dem Testrechner gestartet, erklang nach dem Aufruf der API-Methode `demo` im Verlauf des Startvorgangs wie erwartet ein konstanter Ton mit einer Frequenz von 750 Hz und dem charakteristischen Obertonspektrum einer Sägezahnschwingung ausschließlich auf dem linken Kanal der zum Abhören des Signals benutzten Kopfhörer.[15] Dieser Ton war auch nach Abschluss des Startvorgangs des D3OS, also nachdem das Terminal-Fenster (Abb. 2.1) erschien, weiterhin zu hören und über die gesamte Dauer der Tonwiedergabe waren keine Störgeräusche wie Rauschen oder Knacken wahrnehmbar.[16]
Ergänzend zur Evaluation ohne technische Hilfsmittel über die Ohren wurde das wiedergegebene Signal mit einem Audio-Recorder aufgenommen[17] und das Spektrum dieses Signals mithilfe der freien Open-Source-Tonbearbeitungssoftware Audacity[65] gezeichnet:

[14] Erinnerung: Der Aufruf der Methode `demo` erfolgt über die Referenz auf das vom D3OS instanziierten Struct, welche von der Methode `intel_hd_audio_device` zurückgegeben wird (vgl. 3.2).

[15] Warnung: Kopfhörer sollten beim Testen des Treibers vorsichtshalber nie direkt über den Ohren getragen werden. Fehler in der Audio-Programmierung können sehr laut sein!!!

[16] Rauschen könnte auf das Lesen falscher Speicherbereiche durch die DMA-Engine des *Strems* hindeuten und Knacken auf einen inkorrekten Loop innerhalb der DMA-Bereiche.

[17] Die Aufnahme befindet sich im Repository[20]: D3OS/saw_750hz.wav

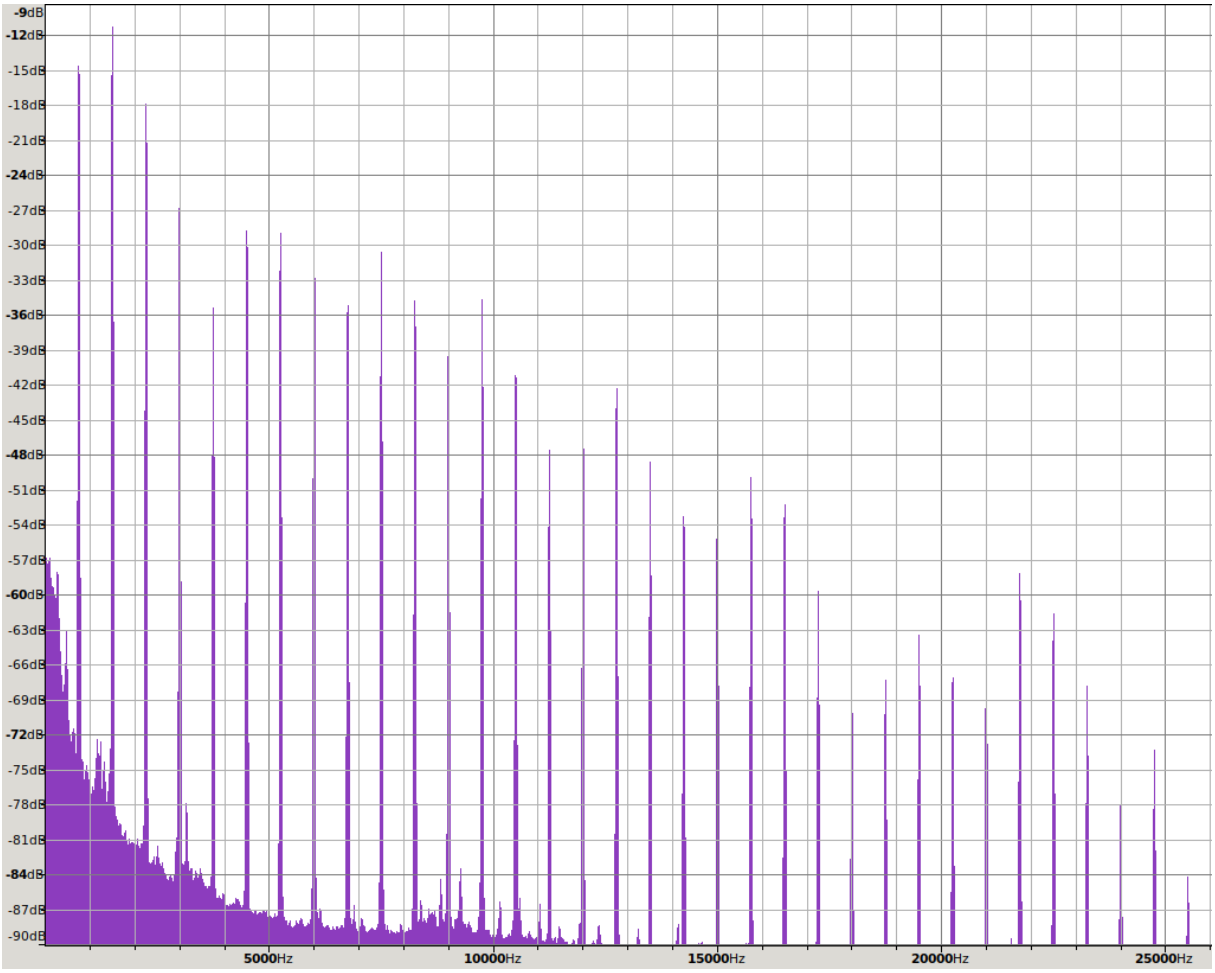


Abbildung 4.1: Spektrum des aufgezeichneten Signals

Die Aufnahme des Signals fand nicht unter Laborbedingungen statt und es wurde keine geeichte Messtechnik benutzt. Dennoch ist das charakteristische Spektrum einer Sägezahnschwingung (vgl. 2.3b) im Spektrum des aufgezeichneten Signals deutlich wiedererkennbar.
Es fällt auf, dass neben den erwarteten Spitzen bei allen Vielfachen der Grundfrequenz von 750 Hz viele Frequenzen zwischen 0 und etwa 10 kHz mit tendenziell abfallender Amplitude im Spektrum enthalten sind. Diese Frequenzen lassen sich durch Nebengeräusche während der Aufnahme erklären, insbesondere durch die Betriebsgeräusche des Testrechners. Außerdem sind im Spektrum in unmittelbarer Nähe jeder einzelnen der erwarteten Spitzen weitere unwartete Spitzen erkennbar. Diese Artefakte können durch einen Aliasing genannten Effekt bei der Digital-Analog-Konvertierung des Signals durch die Soundkarte erklärt und auf die Art und Weise, auf welche die Sägezahnschwingung im Speicher konstruiert wurde, zurückgeführt werden.[18]

[18]Die im Rahmen der Evaluation durch das explizite Festlegen der Werte der einzelnen Samples konstruierte Sägezahnschwingung ist nicht bandbegrenzt und enthält somit Frequenzanteile oberhalb der Frequenz, welche genau der Hälfte der genutzten Samplerate entspricht. Diese Frequenz wird als Nyquist-Frequenz bezeichnet und beträgt im konkreten Fall 24 kHz.[66] Die Frequenzanteile des konstruierten Signals oberhalb der Nyquist-Frequenz werden bei der Digital-Analog-Konvertierung falsch interpretiert und auf Frequenzen unterhalb der Nyquist-Frequenz abgebildet, was als Aliasing bezeichnet wird und durch komplexere bandbegrenzte Konstruktionsverfahren vermieden werden kann.[67]

Da das grundsätzliche Funktionieren der Wiedergabe eines Audiosignals ohne Störgeräusche allerdings bereits ohne technische Hilfsmittel verifiziert werden konnte und es begründete Erklärungen für die unerwarteten Frequenzen innerhalb des aufgezeichneten Spektrums gibt, wurden keine fortführenden Experimente durchgeführt.[19] Da sich außerdem während der Entwicklungsphase gezeigt hat, dass selbst kleine Programmierfehler in der Regel zu sehr drastischen Veränderungen des wiedergegebenen Signals führen, das Spektrum des aufgezeichneten Signals jedoch unverkennbare Ähnlichkeiten mit dem charakteristischen Spektrum einer Sägezahnschwingung mit einer Frequenz von 750 Hz besitzt, wurde gefolgert, dass die Wiedergabe eines *Streams* nicht nur prinzipiell funktioniert, sondern sehr wahrscheinlich auch korrekt ist.

Die erfolgreiche Wiedergabe eines *Streams* über einen analogen Ausgang der Soundkarte nach dem Aufruf der API-Methode `demo` beweist implizit, dass das in 2.6.2 vorgestellte Programmiermodell prinzipiell korrekt implementiert wurde. Außerdem zeigt sie bereits, dass die Soundkarte auf zu drei verschiedenen Zwecken allozierte DMA-Bereiche erfolgreich zugreifen kann, nämlich die *Registers* des *Controllers*, die *Buffer Descriptor List* eines Streams sowie auf die eigentlichen Audiobuffer. Die Wiedergabe beweist jedoch nicht das Funktionieren von *CORB* und *RIRB* sowie des *DMA Position Buffers*, da der Treiber in seiner vorgestellten Version einerseits statt *CORB* und *RIRB* die *ICIO Registers* zur Kommunikation mit den *Codecs* nutzt und der *DMA Position Buffer* andererseits nicht innerhalb der Funktion `demo` ausgelesen wird. Allerdings wurden bei der Instanziierung des Structs `IntelHDAudioDevice` durch das D3OS auch die beiden anderen Testfunktionen, `test_corb_and_rirb` und `test_dma_position_buffer`, aufgerufen und da die Initialisierung der Soundkarte innerhalb dieser Funktionen nicht abgebrochen wurde, ist ebenfalls gezeigt, dass der *Controller* erfolgreich sowohl auf *CORB* und *RIRB* als auch den *DMA Position Buffer* zugreifen kann. Somit sind alle in 2.6.3 aufgelisteten Zwecke durch die Tests abgedeckt.
Da die Funktion `demo` außerdem von außerhalb der Module des Treibers aufgerufen werden konnte, erfüllt das Struct `IntelHDAudioDevice` schließlich auch den in 1.1 geforderten Zweck einer API zur Steuerung einer IHDA-Soundkarte, sodass das Ziel dieser Arbeit insgesamt erfolgreich umgesetzt werden konnte.

[19] Allerdings wurden kleine Variationen der Funktion `demo` mit unterschiedlichen *Stream Formats* und Wellenlängen der Sägezahnschwingung erstellt, jedoch nicht aufgezeichnet. Durch diese Variationen konnten einerseits *Streams*, welche mehr als zwei Audiobuffer oder verschieden große Audiobuffer besaßen und andererseits ein *Stream* mit zwei *Channels* (stereo) erfolgreich wiedergegeben werden. Ändert man im *Stream Format* der vorgestellten Funktion `demo` lediglich die Anzahl der *Channels* von einem auf zwei, müssen sich beide *Channels* die Samples in den Audiobuffern teilen, wodurch auf beiden Kanälen des Kopfhörers jeweils eine Sägezahnschwingung mit einer Frequenz von 1500 Hz zu hören ist.

Kapitel 5

Fazit und Ausblick

Im letzten Kapitel dieser Arbeit werden die vorherigen Kapitel in 5.1 kurz zusammengefasst und anschließend in 5.2 ein Aublick auf mögliche Erweiterungen des Treibers gegeben.

5.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Intel HD Audio Soundkartentreiber entwickelt, der dem D3OS eine API bereitstellt, durch welche dieses Betriebssystem ein im Systemspeicher definiertes digitales Audiosignal erfolgreich über einen analogen Ausgang der integrierten Soundkarte des für die Entwicklung genutzten Testrechners wiedergeben konnte. Für die Implementierung des Treibers wurden Programmierkenntnisse in Rust, allgemeine Kenntnisse über Konzepte der Betriebssystemprogrammierung, insbesondere über virtuelle Speicherverwaltung und DMA, spezifische Kenntnisse über die Umsetzung dieser Konzepte im D3OS, Kenntnisse über den PCI-Bus und das PCI-Interface, Grundkenntnisse in Signalverarbeitung und schließlich sehr detaillierte Kenntnisse über die Intel HD Audio Spezifikation benötigt.
Die durch die Spezifikation vorgegebenen und bereits in 2.6.3 zusammengefassten Konzepte wurden in einem objektorientierten Ansatz in der Sprache Rust umgesetzt und der entwickelte Treiber analog zu bereits existierenden Treibern des D3OS in das Betriebssystem integriert. Die Struktur des Treibers orientiert sich sowohl an dem in 2.6.2 vorgestellten Programmiermodell als auch an den bereits vorhandenen Treibern.
Im Rahmen der Evaluation wurde gezeigt, dass der entwickelte Treiber die in 1.1 an ihn gestellten Anforderungen erfüllt und die Soundkarte des Testrechners außerdem korrekt auf alle allozierten DMA-Bereiche zugreifen konnte. Neben dem Treiber selbst wurden im Rahmen der Evaluation auch die Herausforderungen im Verlauf der Treiberentwicklung betrachtet und beschrieben, insbesondere wie mit der Offenheit und Komplexität der Spezifikation zu Intel HD Audio umgegangen wurde.

5.2 Mögliche Erweiterungen

In seiner Version zum Zeitpunkt der Fertigstellung dieser Arbeit enthält der Code des entwickelten Treibers bereits über 70 Structs und Enums und fast 250 Funktionen und Methoden.

Die erste Zahl wird hauptsächlich durch die im Codec-Modul definierten Repräsentationen aller *Commands*, *Payloads* und *Responses* in die Höhe getrieben, welche über 80 Prozent aller Structs und Enums ausmachen. Dabei wurde sogar etwa die Hälfte aller durch die Spezifikation definierten *Commands*, *Payloads* und *Responses* noch gar nicht implementiert. Diese fehlenden Repräsentationen sollten jedoch leicht nach dem Vorbild der bereits existierenden hinzugefügt werden können.
Den größten Anteil an der zweiten Zahl bilden die Methoden des im Controller-Modul definierten Structs `Controller`. Diese Methoden machen über 60 Prozent aller Funktionen und Methoden aus, wodurch die Tatsache widergespiegelt wird, dass der *Controller* in der IHDA-Architektur die zentrale Schnittstelle zwischen Treiber und Soundkarte darstellt. Die meisten dieser Methoden ermöglichen dabei einen konkreten atomaren Zugriff auf ein bestimmtes *Register* (vgl. 3.3.3). Im Hinblick auf mögliche Erweiterungen wurden während der Entwicklung des Treibers für die meisten *Register* des *Controllers* bereits alle sinnvollen atomaren Methoden implementiert, auch wenn sie von der in dieser Arbeit vorgestellten Version des Treibers noch gar nicht aufgerufen werden. Bevor dem Treiber Funktionalität für den Zugriff auf ein *Register* hinzugefügt wird, sollte also überprüft werden, ob diese Funktionalität bereits auf dem Struct `Controller` implementiert wurde.
In 3.3.4 wurde erklärt, dass der Treiber den *Codec* während der Initialisierung der Soundkarte befragt, um einerseits die Topologie der an den *Link* angeschlossenen *Codecs* zu ermitteln und andererseits möglichst viele statische Informationen über die einzelnen *Nodes* eines *Codecs* zu sammeln. Dieses Sammeln der Informationen wird in der Funktion `scan_function_group_for_available_widgets` durchgeführt und muss noch für ein paar in dieser Arbeit nicht besprochene Typen von *Widgets* implementiert werden.[1]

Einer der nächsten Schritte in der Weiterentwicklung des Treibers sollte die Umstellung des Mechanismus zur Kommunikation mit den *Codecs* von den *ICIO Registers* auf die beiden Ringbuffer *CORB* und *RIRB* sein.[2] Im Rahmen der Evaluation wurde gezeigt, dass diese Buffer bereits betrieben werden können, jedoch existiert, anders als zum Beispiel für Zugriffe auf die *Registers* des *Controllers* über die Methoden des Structs `Register`, noch keine Logik, um komfortabel auf diese Buffer zuzugreifen[3] und Buffer-Überläufe zu verhindern. Zu diesem Zweck könnten in Fortführung des gewählten objektorientierten Ansatzes Repräsentationen der beiden Ringbuffer in Form von Structs definiert werden.
Analog sollte auch das Schreiben in die Audiobuffer eines *Streams* vereinfacht werden[4], indem zunächst ein Dateiformat für PCM-Daten definiert wird. Dieses Dateiformat sollte alle Sampleraten, Bittiefen und Anzahlen an *Channels* unterstützen, die ein *Stream* gemäß der Spezifikation haben kann. Anschließend sollte eine Logik entwickelt werden, welche einerseits in diesem Dateiformat vorliegende Daten in die Audiobuffer schreibt und andererseits Daten aus den Audiobuffern in dieses Dateiformat übersetzt. Wie auch *CORB* und *RIRB* ist der *Cyclic Buffer* eines Streams letztendlich auch ein (auf mehrere Audiobuffer verteilter) Ringbuffer, sodass es womöglich sinnvoll sein könnte, diese Änderungen

[1] Diese *Widgets* heißen *Audio Selector Widget*, *Power Widget*, *Volume Knob Widget* und *Beep Generatoer Widget*. Die ebenfalls in der Spezifikation erwähnten *Vendor Defined Widgets* können nur unterstützt werden, nachdem der jeweilige Hersteller weitere Informationen zu diesen *Widgets* bereitgestellt hat.

[2] In 2.6.1.2 wurde erläutert, warum *CORB* und *RIRB* grundsätzlich zu bervorzugen sind.

[3] Innerhalb der Testfunktion `test_corb_and_rirb` wird momentan noch sehr sperrig über rohe Zeiger auf die einzelnen Buffer-Einträge zugegriffen.

[4] Jedes Sample der durch die Testfunktion `demo` abgespielten Sägezahnschwingung musste manuell in eine PCM-Darstellung gebracht und an die richtige Stell im *Cyclic Buffer* geschrieben werden.

beim Zugriff auf die Audiobuffer in einem Zug mit den eben vorgeschlagenen Änderungen beim Schreiben auf *CORB* und *RIRB* durchzuführen.

Noch besitzt der Treiber keine Funktionalität zur Verarbeitung von *Input-Streams* bzw. zur Audioaufnahme über ein Mikrofon. Die Implementierung dieser Funktionalität sollte sehr ähnlich zur in dieser Arbeit besprochenen Implementierung eines *Output-Streams* durchgeführt werden können, und die bereits existierenden Repräsentationen zur Erstellung eines Streams im Controller-Modul sollten ebenfalls für *Input-Streams* genutzt werden können. Zuvor sollte jedoch das eben angesprochene Dateiformat für PCM-Daten sowie die Logik zur Konvertierung eines *Streams* in dieses Dateiformat entwickelt werden, da das manuelle Zusammensetzen und Speichern der pausenlos über den *Link* einströmenden Samples sonst sehr müßig wäre.

Momentan implementiert der Interrupt-Handler im API-Modul des Treibers noch keine sinnvollen Reaktionen auf Hardware-Interrupts. Beispielsweise können Interrupts gemäß der Spezifikation dazu genutzt werden, während des Betriebs der Soundkarte zu erkennen, wenn ein akustisches Gerät über eine Buchse eines *Codecs* mit der Soundkarte verbunden wird, sodass eine Plug-and-Play-Funktionalität bereitgestellt werden kann. Außerdem kann eingestellt werden, dass ein Interrupt ausgelöst werden soll, sobald eine vorher definierte Anzahl an *Responses* im *RIRB* eingetroffen ist. Die wichtigsten Interrupts sind jedoch die bereits erwähnten *Interrupts on Completion* der Audiobuffer eines *Streams*. Ohne diese *Interrupts on Completion* kann der Treiber nicht wissen, wann er die Daten in einem Audiobuffer austauschen soll, sodass entweder nur repetitive Signale, wie beispielsweise eine Sägezahnschwingung mit konstanter Frequenz, wiedergegeben werden können oder vom Treiber abgeschätzt werden muss, wann die Daten eines Audiobuffers ausgetauscht werden müssen, was sowohl komplizierter, rechenintensiver als auch fehleranfälliger ist als eine Umsetzung über Interrupts.[5]

Gemäß der Spezifikation besitzt ein IHDA-Gerät noch weitere Funktionalität, welche im Rahmen dieser Arbeit nicht diskutiert wurde. Dazu gehören die Synchronisation von *Streams* sowie der *Controller* verschiedener IHDA-Geräte[68], Striping[69], wodurch ein Stream gleichzeitig von mehreren DMA-Engines verarbeitet werden kann, sowie Power Management[70], durch welches verschiedene Zustände zwischen Vollbetrieb und Nichtbetrieb für *Controller* und *Codecs* definiert werden.

Schließlich unterstützt der Treiber in seiner Version zur Fertigstellung dieser Arbeit genau eine Soundkarte, nämlich die integrierte Soundkarte des für die Entwicklung genutzten Testrechners. Um weitere IHDA-Gerät unterstützen zu können, muss die Konfigurierung des *Codecs* (vgl. 2.6.2.5) für jedes Modell individuell angepasst werden[6] (vgl. 4.1). Außerdem wurden im Rahmen dieser Arbeit weder *Modem Codecs* noch *Vendor Defined Codecs* und auch keine *Codecs* mit digitalen Schnittstellen zur Außenwelt wie HDMI, Displayport oder S/PDIF betrachtet (vgl. 3.3.4). Es ist also noch ein weiter Weg zurückzulegen, falls

[5] Bei Signalen mit bekannter Länge kann getrickst werden, indem ein Audiobuffer alloziert wird, in welchen alle abzuspielenden Samples passen. Diese Strategie scheitert jedoch bei Anwendungsfällen, in denen die Länge des abzuspielenden Signals nicht bekannt ist, wie beispielsweise beim Streaming.

[6] Erinnerung: Alle anderen vom Treiber erledigten Schritte des Programmiermodells sollten bereits für beliebige IHDA-Geräte funktionieren.

der Treiber irgendwann alle existierenden IHDA-Geräte unterstützen soll. Und da auch in Zukunft noch neue Geräte nach der IHDA-Spezifikation entwickelt werden dürften, bleibt die Fertigstellung des Treibers wohl eine unendliche Geschichte.

Kapitel 6

Anhang

6.1 Kompilierung und Start des D3OS

Die folgende Anleitung beschreibt, wie das D3OS auf dem Entwicklungsrechner mit installiertem Ubuntu 22.04.4 LTS (Jammy Jellyfish) während der Entwicklung des Treibers kompiliert und anschließend sowohl auf dem physischen Testrechner als auch auf dem virtuellen Testrechner gestartet werden konnte. Es wurde Version 1.78.0-nightly[1] des Rust Compilers und cargo-make Version 0.37.11 benutzt.

Das D3OS wurde durch das Ausführen des Befehls `cargo make --no-workspace` aus dem Wurzelverzeichnis des Repositorys[20] zunächst kompiliert und anschließend in eine Image-Datei[2] geschrieben. Diese Image-Datei wurde einerseits auf einen USB-Stick geschrieben, von dem aus das D3OS auf dem physischen Testrechner gestartet werden konnte, und andererseits wurde die Image-Datei QEMU bei dessen Start als Parameter übergeben, wodurch das D3OS auf dem virtualisierten Testrechner gestartet wurde. Der Aufruf von QEMU wurde allerdings nicht manuell durchgeführt, sondern über ein zu diesem Zweck erstelltes Skript[3], welches durch den Befehl `./run.sh` aus dem Wurzelverzeichnis des Repositorys ausgeführt werden kann. Dieses Skript konfiguriert und startet QEMU so, dass ein Rechner mit eingebauter Intel HD Audio Soundkarte emuliert wird, auf welchem das D3OS von dem erstellten Image aus hochgefahren wird. Es wurde QEMU Version 8.2.1. benutzt.

6.2 Visualisierung der Topologie eines Codecs

In 2.6.1.1 wurde die modulare Architektur eines *Codecs* vorgestellt und in 2.6.2.3 wurde erklärt, dass der Treiber die Topologie eines *Codecs* selbst ermitteln muss. Abb. 6.1 visualisiert die durch den Treiber ermittelte Topologie des einzigen *Codecs* der in den Testrechner verbauten integrierten Soundkarte[4], welcher genau eine *Function Group* ent-

[1] Rust wird über drei Release Channels veröffentlicht: Stable, Beta und Nightly. Zum Entstehungszeitpunkt dieser Arbeit sind einige vom D3OS genutzte „Unstable Features“ von Rust lediglich in den Nightly Releases verfügbar.[71]

[2] Pfad im Repository: D3OS/d3os.img

[3] Pfad im Repository: D3OS/run.sh

[4] Intel Corporation 8 Series/C220 Series Chipset High Definition Audio Controller (rev 04) mit Vendor ID 0x8086 und Device ID 0x8C20

hielt. Jeder Knoten im Verknüpfungsgraphen repräsentiert eines der insgesamt 35 *Widgets* innerhalb dieser *Function Group*, und die Zahlen in den Knoten geben die einzigartige *Node ID* des *Widgets* an.[5] Jeder Pfeil repräsentiert einen Eintrag in der *Connection List* des *Widgets*, von welchem dieser Pfeil ausgeht.
Der *Codec* enthielt die folgenden Typen von *Widgets*:

1. *Audio Output Converter Widget* (*AOCW*) und *Audio Input Converter Widget* (*AICW*), welche immer direkt mit dem Link verbunden sind.

2. *Mixer Widget* (*MW*) und *Selector Widget* (*SW*), welche für das Routing eines Audiosignals genutzt werden.

3. *Line Out Pin Widget* (*LOPW*), *Speaker Pin Widget* (*SPW*), *HP Out Pin Widget* (*HPOPW*), *SPDIF Out Pin Widget* (*SPDIFOPW*) und *Mic In Pin Widget* (*MIPW*), welche immer direkt jeweils mit genau einer Buchse des *Codecs* verbunden sind.

4. Zuletzt enthielt der *Codec* ein Dutzend *Vendor Defined Widgets* (*VDW*), welche jeweils mit keinem anderen *Widget* verbunden waren und deren Funktion bis zum Ende der Entwicklungsphase ungeklärt blieb.

In 2.6.2.5 wurde erläutert, dass der Treiber zunächst einen Ausgabepfad innerhalb der Topologie des *Codecs* finden und die einzelnen *Widgets* auf diesem Ausgabepfad individuell konfigurieren muss, bevor ein *Stream* über einen analogen Ausgang der Soundkarte wiedergegeben werden kann.
Der durch den Treiber gefundene Pfad ist im Diagramm blau eingefärbt, wobei das über den *Link* erhaltene und vom *AOCW* konvertierte Signal über das *MW* entgegen der Pfeilrichtungen zum *LOPW* fließt.
In 4.1 wurde außerdem erwähnt, dass beim Suchen eines Ausgabepfades auf Zyklen im Verknüpfungsgraphen geachtet werden muss. Beispielsweise enthält der Verknüpfungsgraph einen Zyklus vom *MW* auf dem gefundenen Ausgabepfad über zwei andere *Widgets* außerhalb des Ausgabepfades zurück zum *MW*. Dieser Zyklus wurde rot eingefärbt.

[5] Die *Node IDs* fangen bei der Zahl 2 an, da die *Node ID* 0 für die *Root Node* des *Codecs* reserviert ist und die *Node ID* 1 zur einzigen *Function Group Node* des *Codecs gehört*. Diese beiden *Nodes* sind nicht im Verknüpfungsgraphen der *Widgets* enthalten.

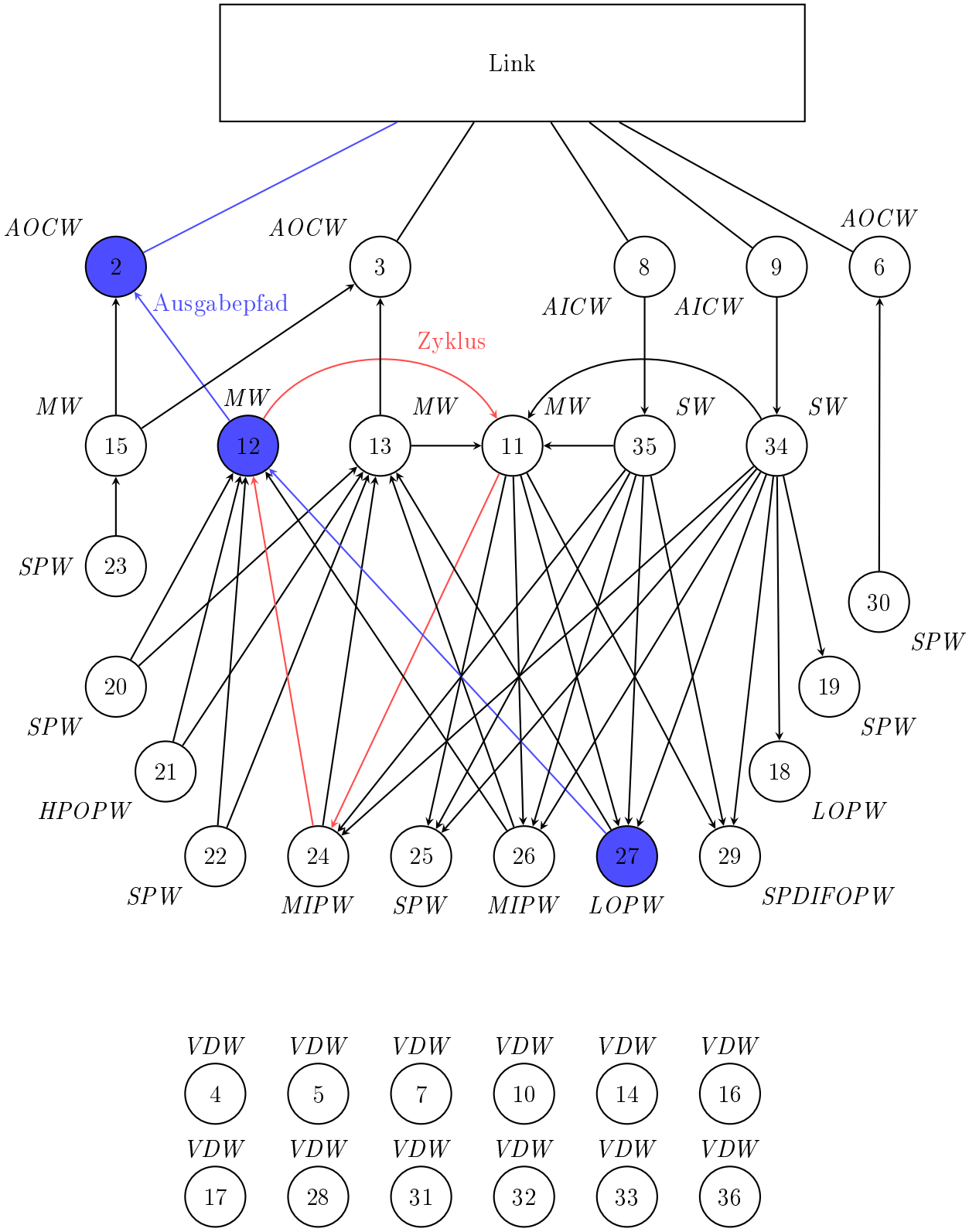


Abbildung 6.1: Verknüpfungsgraph der *Widgets* innerhalb des *Codecs* der in den Testrechner verbauten integrierten Soundkarte

Literatur

[1] "Logical Components of the Host", in *Information Storage and Management: Storing, Managing, and Protecting Digital Information*. Indianapolis: Wiley Publishing, Inc., 2009, S. 39, ISBN: 978-0-470-29421-5.

[2] Intel Corporation, *High Definition Audio Specification Revision 1.0a*, 17.06.2010, PDF verfügbar unter `https://www.intel.com/content/www/us/en/standards/high-definition-audio-specification.html`, besucht am 16.05.2024.

[3] The Rust Programming Language Blog, *Announcing Rust 1.0*, 15.05.2015, `https://blog.rust-lang.org/2015/05/15/Rust-1.0.html`, besucht am 16.05.2024.

[4] Rust Team, *Community*, `https://www.rust-lang.org/community`, besucht am 16.05.2024.

[5] Rust Team, *Production Users*, `https://www.rust-lang.org/production/users`, besucht am 16.05.2024.

[6] S. Klabnik und C. Nichols mit Beiträgen der Rust Community, *The Rust Programming Language*, 2. Aufl. 2023, `https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`, besucht am 16.05.2024.

[7] Y. Yakimenko, *Checklist: Nine most frequent memory allocation errors in C/C++ made by developers*, `https://www.hoist-point.com/most-frequent-memory-errors-in-cpp.htm`, besucht am 16.05.2024.

[8] S. Klabnik und C. Nichols mit Beiträgen der Rust Community, *The Rust Programming Language*, 2. Aufl. 2023, `https://doc.rust-lang.org/book/ch00-00-introduction.html#people-who-value-speed-and-stability`, besucht am 16.05.2024.

[9] A. Avram, *Interview on Rust, a Systems Programming Language Developed by Mozilla*, `https://www.infoq.com/news/2012/08/Interview-Rust`, besucht am 16.05.2024.

[10] Rust Team, *Unsafety Checking*, `https://rustc-dev-guide.rust-lang.org/unsafety-checking.html`, besucht am 16.05.2024.

[11] S. Klabnik und C. Nichols mit Beiträgen der Rust Community, *The Rust Programming Language*, 2. Aufl. 2023, `https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html#dereferencing-a-raw-pointer`, besucht am 16.05.2024.

[12] S. Klabnik und C. Nichols mit Beiträgen der Rust Community, *The Rust Programming Language*, 2. Aufl. 2023, `https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html#unsafe-superpowers`, besucht am 16.05.2024.

[13] S. Klabnik und C. Nichols mit Beiträgen der Rust Community, *The Rust Programming Language*, 2. Aufl. 2023, https://doc.rust-lang.org/book/ch07-00-managing-growing-projects-with-packages-crates-and-modules.html, besucht am 16.05.2024.

[14] S. Klabnik und C. Nichols mit Beiträgen der Rust Community, *The Rust Programming Language*, 2. Aufl. 2023, https://doc.rust-lang.org/book/ch01-03-hello-cargo.html, besucht am 16.05.2024.

[15] S. Gur-Ari, *cargo-make*, https://crates.io/crates/cargo-make, 2010, besucht am 16.05.2024.

[16] Abteilung Betriebssysteme der Heinrich-Heine-Universität Düsseldorf, *D3OS*, https://www.cs.hhu.de/en/research-groups/operating-systems/research/translate-to-english-hermes-1, besucht am 16.05.2024.

[17] P. Oppermann, *Writing an OS in Rust (Philipp Oppermann's blog)*, https://os.phil-opp.com/, besucht am 16.05.2024.

[18] Abteilung Betriebssysteme der Heinrich-Heine-Universität Düsseldorf, *D3OS*, https://github.com/hhu-bsinfo/D3OS, besucht am 16.05.2024.

[19] Software Freedom Conservancy, *QEMU*, https://www.qemu.org, besucht am 16.05.2024.

[20] Abteilung Betriebssysteme der Heinrich-Heine-Universität Düsseldorf und S. Heidelberg, *D3OS*, https://github.com/sehei107/D3OS, besucht am 16.05.2024.

[21] Autoren des OSDev Wiki, *The PCI Bus*, https://wiki.osdev.org/PCI#The_PCI_Bus, besucht am 16.05.2024.

[22] Autoren des OSDev Wiki, *PCI Configuration Space*, https://wiki.osdev.org/PCI#Configuration_Space, besucht am 16.05.2024.

[23] "Pulse-Code-Modulation, digitale Signalverarbeitung und Audio-Codierung", in *Nachrichtentechnik: Eine Einführung für alle Studiengänge*. Wiesbaden: Vieweg+Teubner, 2006, S. 67–96, ISBN: 978-3-8348-9097-9. DOI: 10.1007/978-3-8348-9097-9_3. Adresse: https://doi.org/10.1007/978-3-8348-9097-9_3.

[24] *Nachrichtentechnik: Eine Einführung für alle Studiengänge*, Bild 3-10, S. 68.

[25] J. Wilczek, *Sine, Saw, Square, Triangle, Pulse: Basic Waveforms in Synthesis and Their Properties*, https://thewolfsound.com/sine-saw-square-triangle-pulse-basic-waveforms-in-synthesis, besucht am 16.05.2024.

[26] *High Definition Audio Specification Revision 1.0a*, ch. 2.1 Hardware System Overview, p. 18–19.

[27] *High Definition Audio Specification Revision 1.0a*, Figure 1. High Definition Audio Architecture Block Diagram, p. 18.

[28] *High Definition Audio Specification Revision 1.0a*, ch. 3 Register Interface, p. 24–52.

[29] *High Definition Audio Specification Revision 1.0a*, ch. 3.3.2 Offset 00h: GCAP – Global Capabilities, p. 28.

[30] *High Definition Audio Specification Revision 1.0a*, ch. 7 Codec Features and Requirements, p. 127.

[31] *High Definition Audio Specification Revision 1.0a*, ch. 1.2.2 Feature List, p. 17.

[32] *High Definition Audio Specification Revision 1.0a*, ch. 7.1.1 Modular Architecture, p. 127–128.

[33] *High Definition Audio Specification Revision 1.0a*, ch. 7.2 Qualitative Node Definition, p. 131–139.

[34] *High Definition Audio Specification Revision 1.0a*, ch. 7.3.4 Parameters, p. 198–214.

[35] *High Definition Audio Specification Revision 1.0a*, ch. 7.3.3 Controls, p. 141–198.

[36] *High Definition Audio Specification Revision 1.0a*, Figure 50. Codec Module Addressing Scheme, p. 129.

[37] *High Definition Audio Specification Revision 1.0a*, ch. 7.3 Codec Parameters and Controls, p. 139.

[38] *High Definition Audio Specification Revision 1.0a*, ch. 5.3.1 Basic Frame Components, p. 83–84.

[39] *High Definition Audio Specification Revision 1.0a*, ch. 3.7 Codec Verb and Response Structures, p. 57.

[40] *High Definition Audio Specification Revision 1.0a*, ch. 4.4.1 Command Outbound Ring Buffer – CORB, p. 62–66.

[41] *High Definition Audio Specification Revision 1.0a*, ch. 4.4.2 Response Inbound Ring Buffer - RIRB, p. 66–69.

[42] *High Definition Audio Specification Revision 1.0a*, ch. 3.4 Immediate Command Input and Output Registers, p. 50–52.

[43] *High Definition Audio Specification Revision 1.0a*, ch. 2.2 Streams and Channels, p. 19–21.

[44] Intel Corporation, *HIGH DEFINITION AUDIO ("AZALIA") SPECIFICATION DEVELOPMENT AGREEMENT*, `https://www.intel.com/content/dam/www/public/us/en/documents/license-agreements/high-definition-audio-specification-development-agreement.pdf`, besucht am 16.05.2024.

[45] *High Definition Audio Specification Revision 1.0a*, Figure 2. Streams, p. 20.

[46] *High Definition Audio Specification Revision 1.0a*, ch. 3.7.1 Stream Format Structure, p. 58–59.

[47] *High Definition Audio Specification Revision 1.0a*, ch. 4.5.1 Stream Data In Memory, p. 69–70.

[48] *High Definition Audio Specification Revision 1.0a*, ch. 4.5.3 Starting Streams, p. 70–71.

[49] *High Definition Audio Specification Revision 1.0a*, ch. 4.5.4 Stopping Streams, p. 71.

[50] *High Definition Audio Specification Revision 1.0a*, ch. 3.6.1 DMA Position in Current Buffer, p. 55.

[51] Autoren des OSDev Wiki, *Identifying HDA on a machine*, `https://wiki.osdev.org/Intel_High_Definition_Audio#Identifying_HDA_on_a_machine`, besucht am 16.05.2024.

[52] Autoren des OSDev Wiki, *PCI Command Register*, `https://wiki.osdev.org/PCI#Command_Register`, besucht am 16.05.2024.

[53] Autoren des OSDev Wiki, *PCI IRQ Handling*, `https://wiki.osdev.org/PCI#IRQ_Handling`, besucht am 16.05.2024.

[54] Autoren des OSDev Wiki, *PCI Device Registers*, `https://wiki.osdev.org/Intel_High_Definition_Audio#Device_Registers`, besucht am 16.05.2024.

[55] *High Definition Audio Specification Revision 1.0a*, ch. 4.2.2 Starting the High Definition Audio Controller, p. 61.

[56] *High Definition Audio Specification Revision 1.0a*, ch. 4.3 Codec Discovery, p. 62.

[57] *High Definition Audio Specification Revision 1.0a*, ch. 3.3.7 Offset 08h: GCTL – Global Control, p. 30.

[58] *High Definition Audio Specification Revision 1.0a*, ch. 3.3.14 Offset 20h: INTCTL – Interrupt Control, p. 34.

[59] *High Definition Audio Specification Revision 1.0a*, ch. 7.1 Codec Architecture, p. 127.

[60] *High Definition Audio Specification Revision 1.0a*, ch. 7.3.4.6 Audio Widget Capabilities, p. 201–204.

[61] *High Definition Audio Specification Revision 1.0a*, ch. 7.3.3.3 Get Connection List Entry, p. 142–143.

[62] Autoren des OSDev Wiki, *Setting up the AFG codec*, `https://wiki.osdev.org/Intel_High_Definition_Audio#Setting_up_the_AFG_codec`, besucht am 16.05.2024.

[63] Linux developers, *hda_intel.c*, `https://github.com/torvalds/linux/blob/master/sound/pci/hda/hda_intel.c`, besucht am 16.05.2024.

[64] RedoxOS developers, *redox-os/drivers/audio/ihdad*, `https://gitlab.redox-os.org/redox-os/drivers/-/tree/master/audio/ihdad`, besucht am 16.05.2024.

[65] The Audacity Team, *Audacity*, `https://www.audacityteam.org`, besucht am 16.05.2024.

[66] "Nyquist-Bandbreite und Impulsformung", in *Nachrichtentechnik: Eine Einführung für alle Studiengänge*. Wiesbaden: Vieweg+Teubner, 2006, S. 144–148, ISBN: 978-3-8348-9097-9. DOI: 10.1007/978-3-8348-9097-9_5. Adresse: `https://doi.org/10.1007/978-3-8348-9097-9_5`.

[67] T. Stilson und J. Smith, *Alias-Free Digital Synthesis of Classic Analog Waveforms*, `https://ccrma.stanford.edu/~stilti/papers/blit.pdf`, besucht am 16.05.2024.

[68] *High Definition Audio Specification Revision 1.0a*, ch. 4.5.7 Synchronization, p. 72–73.

[69] *High Definition Audio Specification Revision 1.0a*, ch. 5.3.2.3 Outbound Frame Overview – Multiple SDO, p. 87.

[70] *High Definition Audio Specification Revision 1.0a*, ch. 4.5.8 Power Management, p. 73.

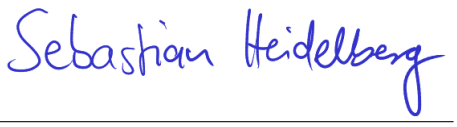
[71] Rust Team, *Unstable features*, `https://doc.rust-lang.org/rustdoc/unstable-features.html`, besucht am 16.05.2024.

Abbildungsverzeichnis

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 18. Mai 2024



Sebastian Heidelberg